UNIVERSIDAD CARLOS III DE MADRID

USO DE TÉCNICAS DE PARALELIZACIÓN PARA EL ALGORITMO DE LOS FILTROS DE KALMAN

Trabajo Fin de Grado

Septiembre de 2012

Autor: Javier Rodríguez Arroyo
Tutor: Luis Miguel Sánchez García
Co-tutor: Javier Fernández Muñoz

AGRADECIMIENTOS

A mi familia, por el apoyo y la comprensión que me han dado tanto en la realización del proyecto como durante toda la carrera.

A Almudena, por estar ahí siempre, incondicionalmente. Sin ti nunca hubiese llegado hasta aquí.

A mis tutores, *Luismi* y *Javi*, por guiarme durante el proyecto y dedicarle tanto tiempo y esfuerzo.

A José Daniel, por ayudarme tanto con C++ y por la confianza que ha depositado en mí durante la carrera.

Por último, aunque no menos importante, a mis compañeros de beca, por los cafés de media tarde sin los que no hubiésemos aguantado despiertos.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCIÓN

En este capítulo se hace una introducción al tema sobre el que trata el proyecto y se explica la problemática que lo motiva así como los objetivos a los que se pretende llegar. También se detalla la estructura del resto del documento.

1.1 DESCRIPCIÓN

El filtro de Kalman es un algoritmo matricial iterativo que resuelve el problema del filtrado lineal de datos discretos en sistemas ruidosos. Su funcionamiento consta de dos etapas, una de predicción de la medida y otra de corrección. En la actualidad es utilizado en multitud de disciplinas.

Una de las aplicaciones del filtro de Kalman es la predicción de la posición de objetos en movimiento. Durante el proyecto se desarrollará un sistema capaz de aplicar este algoritmo al mayor número de objetos móviles posible.

Se debe tener en cuenta que se necesitaran un número mínimo de iteraciones por unidad de tiempo para conseguir una precisión suficiente en las estimaciones. Esto hace que el sistema, además de tener una lógica correcta, debe ser capaz de ejecutar esa lógica en un periodo de tiempo determinado, siendo por lo tanto un sistema de tiempo real. Por otra parte cabe destacar que no es necesario que el sistema cumpla los tiempos de forma estricta, ya que el no cumplimiento de estos no provoca un mal funcionamiento del sistema, sino simplemente un funcionamiento degradado en el cual se verá afectada la precisión en las predicciones, por lo que se puede considerar un sistema de tiempo real no crítico o de calidad de servicio.

El número de objetos cuya posición y velocidad debe predecir el sistema tiene que ser el máximo posible, teniendo en cuenta las limitaciones hardware de la máquina donde se ejecute. Debido a que en la actualidad los computadores modernos son máquinas multicore o manycore, la única forma de optimizar el uso del hardware es a través de la paralelización de los algoritmos, por lo que uno de los pilares del sistema a implementar es que sea lo más concurrente y escalable posible y que aproveche al máximo las capacidades de computo del hardware.

Son muchos los modelos de programación paralela que podemos encontrar en la actualidad. Durante la realización del proyecto nos centraremos en 4 de ellos:

- Intel Threading Building Blocks (TBB).
- Intel Array Building Blocks (ArBB).
- OpenMP.
- Modelo de hilos de C++.

Los tres primeros se implementan con bibliotecas, por lo que son totalmente portables. Además son Open Source.

Debido a la cantidad de arquitecturas diferentes que se pueden encontrar en el mercado, se crea la necesidad de hacer código lo más portable y estándar posible. Por ello, se ha decidido utilizar como lenguaje de programación para el proyecto C++. Este lenguaje es multiparadigma, soporta orientación a objetos y programación genérica. Además actualmente existe un estándar, denominado ISO C++, al que se han adherido la mayoría de los fabricantes de compiladores más modernos. Este estándar ha sido actualizado en año el 2011 añadiendo gran cantidad de funcionalidad al lenguaje. Por otra parte las 3 bibliotecas de paralelización que se van a utilizar tienen soporte para este lenguaje.

1.2 PROBLEMÁTICA

El proyecto enfrenta el problema de crear un sistema en tiempo real de calidad de servicio, capaz de aplicar el algoritmo del filtro de Kalman discreto al mayor número objetos móviles posibles, a partir de los datos que se reciben de varios sensores que monitorizan y toman medidas de cada uno de ellos. Esta predicción debe tener la suficiente precisión.

Uno de los problemas que presenta la implementación es que al ser un algoritmo iterativo hay que seguir un orden al tratar los datos, más concretamente un orden temporal, es decir sólo se deben tratar los datos generados en el instante *t-1* antes de haber tratado los datos generados en el instante *t*. Sin embargo debido a que se deben de tratar varios objetos al mismo tiempo y que estos están monitorizados por varios sensores con distintas características no se puede asegurar que los datos se recibirán en el orden correcto. En el presente proyecto se considerará que los datos se reciben en orden correcto o que se ordenan previamente. Por otra parte se debe asegurar que se tratan todas las medidas de todos los objetos por igual, teniendo que realizar un reparto equitativo de la carga.

Para poder tratar la máxima cantidad de datos por unidad de tiempo es necesario hacer el máximo uso posible del hardware del computador donde se ejecute el software. Para

conseguir el máximo uso de los recursos del sistema se debe hacer una implementación paralela del algoritmo, además esta implementación debe ser lo más escalable posible.

La implementación paralela de un algoritmo implica la posibilidad de que ocurran ciertos fenómenos indeseados como las condiciones de carrera. Estos fenómenos obligan a tener que tomar medidas que habitualmente suele ser el uso de técnicas de exclusión mutua, las cuales producen bloqueos y por lo tanto desperdicios del tiempo de procesador. Este será uno de los problemas a minimizar en la medida de lo posible durante el diseño del sistema, ya que una mala gestión del acceso a memoria puede reducir el rendimiento del sistema en gran medida.

Otra de las dificultades que deben superarse es como se gestiona la memoria en términos de acceso a la misma, ya que el uso de matrices hace que el acceso a memoria no sea tan trivial como en el caso de valores escalares. Además hay que tener en cuenta que aunque no será lo común en este proyecto, el tamaño de la matrices puede llegar a ser muy grande, por lo que se debe encontrar un forma de minimizar en la medida de lo posible el movimiento de datos en memoria, ya que aunque las prestaciones de esta ha avanzado mucho en los últimos años aún sigue siendo muy lentas respecto a la velocidad de los procesadores.

Por último hay que tener en cuenta que la implementación debe ser portable para permitir su ejecución en el mayor número de sistemas posible, por lo que se deberá utilizar un lenguaje de programación estandarizado y que a su vez sea eficiente y aproveche los recursos hardware de la máquina.

1.3 OBJETIVOS

El objetivo principal del proyecto es realizar el análisis, el diseño y la implementación en C++ de un sistema en tiempo real de calidad de servicio para la aplicación del algoritmo del filtro de Kalman lineal o discreto a varios objetos. La consecución de este propósito se puede desglosar en los siguientes subobjetivos:

- Estudio del funcionamiento del algoritmo del filtro de Kalman discreto.
- Familiarización con el lenguaje de programación C++ y con el estándar ISO/IEC C++ 2011.
- Implementación de funciones para realizar operaciones con matrices:
  - Suma de matrices.
  - Resta de matrices.
  - Producto de matrices.
  - Inversa de matrices.
  - Traspuesta de matrices.
  - Copia de matrices.
- Implementación de funciones para llevar a cabo la ejecución del algoritmo del filtro de Kalman discreto:
  - Etapa de predicción.
  - Etapa de corrección.
- Diseño e implementación de un sistema de gestión de los datos que se reciben de los sensores.
- Diseño e implementación de un sistema que simule el envío de datos de los sensores en tiempo real.
- Aprendizaje de modelos de modelos de programación paralela:
  - OpenMP.
  - Intel Threading Building Blocks con tareas.
  - Intel Threading Building Blocks con *pipeline*.
  - Intel Array Building Blocks.
  - Modelo de hilos de C++.
- Implementación del algoritmo de forma paralela con los diferentes modelos de programación.

- Diseño y generación de casos de prueba de rendimiento de las diferentes implementaciones con diferentes parámetros.
- Evaluación del rendimiento de las diferentes implementaciones.

1.4 ESTRUCTURA DEL DOCUMENTO

Esta memoria está organizada en 6 capítulos, el primero de ellos, el actual, proporciona una visión general del proyecto, su problemática y los objetivos que se persiguen.

En el segundo capítulo, *Estado de la cuestión*, se da la base matemática del algoritmo del filtro de Kalman discreto. Además, se explican las diferentes opciones de paralelización existentes en la actualidad, las herramientas utilizadas para medir el rendimiento y una visión general del estándar ISO/IEC C++ 2011.

En el tercer capítulo, *Metodología*, se realiza una enumeración y resumen de las diferentes etapas que se ha llevado a cabo para realizar el proyecto.

En el cuarto capítulo, *Análisis y diseño,* se aborda el análisis, diseño e implementación de las diferentes versiones desarrolladas tanto secuenciales como paralelas.

En el quinto capítulo, *Desarrollo*, se explica detalladamente las diferentes versiones desarrolladas.

En el sexto capítulo, *Evaluación*, se presentan una evaluación del rendimiento del sistema implementado en base a las los resultados obtenidos tras la ejecución de las pruebas de rendimiento realizadas.

En el séptimo y último capítulo, *Conclusiones*, se recogen las conclusiones del proyecto tanto desde el punto de vista técnico como personal.

2 ESTADO DE LA CUESTIÓN

2.1 FILTRO DE KALMAN

En 1960 Rudolf Emil Kalman publicó un artículo describiendo una solución recursiva al problema del filtrado lineal de datos discretos [1]. Desde entonces, y debido en gran parte a los avances en computación, el filtro de Kalman ha sido objeto de numerosas investigaciones y aplicaciones en diversos campos, como por ejemplo:

- **Aviónica:** Aviación autónoma no tripulada. Pilotos automáticos.
- **Astronomía:** Predicción de movimientos.
- **Sistemas de control:** Sistema global de navegación por satélite (GPS).
- **Economía:** Estimación de los parámetros de modelos econométricos.
- **Inteligencia Artificial:** Aprendizaje de redes neuronales multicapa.

El filtro de Kalman es un procedimiento matemático que opera por medio de un mecanismo de predicción y corrección. En esencia este algoritmo pronostica el nuevo estado a partir de su estimación previa, añadiendo un término de corrección proporcional al error de predicción, de tal forma que este último es minimizado estadísticamente.

Existen diferentes variantes del filtro de Kalman dependiendo del problema que se modele. Entre ellas podemos encontrar:

- **Filtro de Kalman discreto o lineal (KF):** Permite estimar el estado $x \in \Re^n$en procesos de tiempo discreto gobernados por una ecuación estocástica lineal [2].
- **Filtro de Kalman Extendido (EKF):** Permite estimar el estado $x \in \Re^n$en procesos de tiempo discreto gobernados por una ecuación diferencial estocástica no lineal [2].
- **Filtro de Kalman "Unscented" (UKF):** Se puede considerar el resultado de incorporar la transformada unscented al EKF para mejorar las aproximaciones que se hacen de los dos primeros momentos de una variable aleatoria que resulta de propagar otra variable aleatoria (supuesta gaussiana) a través de una transformación no lineal [3]. Un estudio detallado de esta variante se puede encontrar en [4].
- **Filtro de Kalman-Bucy:** Es una versión continua del Filtro de Kalman.

Este proyecto se centra en el estudio de la paralelización del filtro de Kalman discreto o lineal. En la mayoría de los casos las operaciones que se realizan en la aplicación del algoritmo del filtro de Kalman son operaciones matriciales, por lo que se prestan a la paralelización de forma que se consiga un alto rendimiento en su aplicación.

Como se ha dicho anteriormente el filtro de Kalman discreto trata el problema de la estimación de un estado $x \in \Re^n$ en un proceso de tiempo discreto gobernado por una ecuación estocástica lineal

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \quad [2.1]$$

Con una medida $z \in \Re^m$ se tiene

$$z_k = Hx_k + v_k \quad [2.2]$$

En la ecuación [2.1] A es una matriz cuadrada $n \times n$ que relaciona el estado en instante anterior $k-1$ con el estado en el instante de tiempo actual k. La matriz B $n \times l$ no siempre existe y está relacionada con el control de entrada. La matriz H en la ecuación [2.2] es una matriz $m \times n$ que describe cómo las mediciones se relacionan con el estado.

Tanto v_k como w_k son variables aleatorias que representan el ruido de la medida y del proceso respectivamente. Estos ruido son independientes uno de otro, blancos y se distribuyen como una normal.

$$p(w) \sim N(0, Q) \quad [2.3]$$
$$p(c) \sim N(0, R) \quad [2.4]$$

Siendo Q la matriz de covarianzas del ruido del proceso y R la matriz de covarianzas del ruido de la medida.

Si se define $\hat{x}_k^- \in \Re^n$ como la estimación del estado a priori en el instante k dado un conocimiento del proceso previo al paso k y se define $\hat{x}_k \in \Re^n$ como la estimación del estado a posteriori en el instante k dada una medida z_k, podemos definir los errores a priori y a posteriori como:

$$e_k^- \equiv x_k - \hat{x}_k^- \quad [2.5]$$

$$e_k \equiv x_k - \hat{x}_k \quad [2.6]$$

Y por tanto sus covarianzas serán su error cuadrático medio a priori:

$$P_k^- = E[e_k^- e_k^{-T}] \quad [2.7]$$

Y a posteriori

$$P_k = E[e_k e_k^T] \quad [2.8]$$

2.1.1 ALGORITMO DEL FILTRO DE KALMAN DISCRETO O LINEAL

El filtro de Kalman estima un proceso utilizando la retroalimentación que le devuelve éste en forma de medidas ruidosas. Las ecuaciones del filtro de Kalman se dividen en dos grupos que forman las dos fases del algoritmo. El primero de ellos lo forman las ecuaciones de predicción que definen la primera fase del algoritmo en la que se obtiene una estimación a priori del siguiente paso o instante. El segundo grupo lo forman las ecuaciones de medida o corrección. En esta fase se introducen las medidas (ruidosas) en la estimación a priori para obtener una estimación a posteriori mejorada.

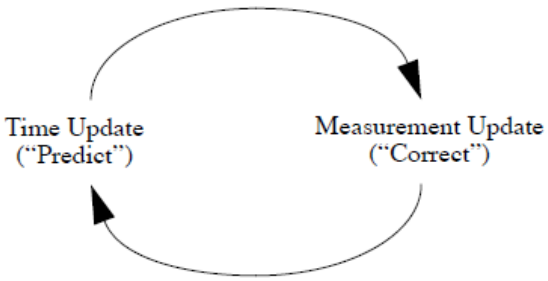

Figura 2.1. Etapas del filtro de Kalman. Fuente: [2].

Las ecuaciones que forman cada etapa del algoritmo son las siguientes:

Etapa de predicción:

$$\hat{x}_k^- = A\hat{x}_{k-1} + B\hat{x}_{k-1} \quad [2.9]$$

$$P_k^- = AP_{k-1}A^T + Q \quad [2.10]$$

Estas dos ecuaciones calculan una estimación del siguiente estado, $\hat{x}_k^-$ y el siguiente valor de covarianzas, P_k^-.

Etapa de corrección:

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} \quad [2.11]$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \quad [2.12]$$

$$P_k = (I - K_k H)P_k^- \quad [2.13]$$

La primera tarea de la etapa de corrección es calcular la ganancia de Kalman, K_k. A continuación con la ecuación [2.12] se actualiza la predicción a priori con la medida, z_k, para obtener la predicción a posteriori, $\hat{x}_k$. Por último se obtiene una estimación de la covarianza a posteriori, P_k, aplicando la ecuación [2.13] del grupo.

2.2 SISTEMAS EN TIEMPO REAL

Un sistema de tiempo real es un sistema informático que cumple que, para que el funcionamiento del mismo sea correcto, no basta con que realice las acciones correctas sino que además estas deben ejecutarse en un intervalo de tiempo especificado

Típicamente un sistema de tiempo real está compuesto de subsistemas que controlan (el computador de control) y subsistemas controlados (el ambiente físico) y está caracterizado por la necesidad de restricciones de tiempo. Las interacciones entre los dos subsistemas están descritas por tres operaciones: muestreo, procesamiento y respuesta. Los subsistemas de computación continuamente procesan los datos muestreados del ambiente físico y producen una respuesta apropiada que es enviada al ambiente físico. Las tres operaciones deben ser realizadas dentro de tiempos específicos. Esto es lo que constituye las restricciones de tiempo. Si una reacción ocurre demasiado tarde podría ser peligrosa para el sistema. Hoy en día, la computación en tiempo real juega un papel importante en nuestra sociedad, puesto que un gran número de sistemas complejos depende en parte o completamente del control por computadora.

Los sistemas en tiempo real se pueden clasificar en dos grandes grupos, sistemas críticos y acríticos.

Sistemas críticos (hard real-time systems)	**Sistemas acríticos (soft real-time systems)**
Plazo de respuesta estricto	Plazo de respuesta flexible
Comportamiento temporal determinado por el entorno	Comportamiento temporal terminado por el computador
Comportamiento en sobrecargas predecible	Comportamiento en sobrecargas degradado
Requisitos de seguridad críticos	Requisitos de seguridad acríticos
Redundancia activa	Recuperación de fallos
Volumen de datos reducido	Gran volumen de datos

Tabla 2.1. Características de sistemas críticos y acríticos.

2.2.1 SISTEMAS CRÍTICOS

Los sistemas de tiempo real crítico son aquellos capaces de ejecutar tareas en un plazo de tiempo muy estricto y cuyo fallo, en términos de plazo temporal, puede conllevar catastróficas consecuencias para el sistema. Los sistemas de tiempo real críticos trabajan en entornos en los que en caso de fallo no solo se tendrán pérdidas económicas sino también humanas, tales como sistemas de control de vuelo, monitorización de pacientes, etc.

En los sistemas de tiempo real críticos se debe garantizar que todas las tareas se cumplen en un plazo de tiempo determinado, por ello estos sistemas están sujetos a grandes restricciones en su ejecución, como el hardware que utilizan, la cantidad de datos que pueden manejar, etc. Todos los factores que componen el entorno de uno de estos sistemas deben estar perfectamente controlados y es necesario caracterizar con precisión la carga máxima y los posibles fallos, además de probar analíticamente su funcionamiento en todos los casos para los que se diseñe.

Dentro de los sistemas críticos podemos encontrar dos subtipos dependiendo de cómo actúan en caso de fallo parcial del sistema. El primer subtipo enmarca aquellos sistemas capaces llevar a cabo una parada en condiciones de seguridad para las personas o cosas de las que el sistema es responsable. Estos sistemas se conocen como sistemas con parada segura y se utilizan en entornos en los que en caso de fallo la parada del sistema evita que ocurra un accidente como por ejemplo sistemas ferroviarios en los que en caso de fallo es posible frenar un tren y esperar a que el sistema se restablezca.

El segundo subtipo engloba los sistemas en los que no es posible realizar una parada segura del sistema debido a los requerimientos del entorno. Estos sistemas suelen tener una gran redundancia y capacidades de funcionamiento en modo degradado ya que no es posible parar el sistema si se produce un funcionamiento incorrecto parcial del mismo. El ejemplo más habitual de este tipo son los sistemas aéreos. En estos sistemas es imposible parar si en mitad de un vuelo parte de las funciones no funcionan correctamente y, por lo tanto, el sistema deberá poder utilizar elementos redundantes que sustituyan a los que no funcionan bien o funcionar en modo degradado hasta llegar a poder realizar una parada segura, en este caso llegar a un aeropuerto para poder llevar a cabo un aterrizaje de emergencia.

2.2.2 SISTEMAS ACRÍTICOS

Los sistemas de tiempo real acríticos son aquellos en los cuales los requisitos temporales tienen cierto grado de flexibilidad. Esta flexibilidad es la característica principal que distingue estos sistemas de los sistemas críticos. Los sistemas acríticos trabajan en entornos en los cuales el fallo del sistema, ya sea total o parcial, no producirá pérdidas importantes y en ningún caso pondrá en peligro vidas humanas. Una de las aplicaciones más comunes de estos sistemas son los sistemas multimedia, en los cuales se requiere una cierta calidad de servicio como una tasa de bits más o menos constante y, que por tanto tienen requisitos temporales pero sin que su fallo ponga a nada ni nadie en peligro.

Los sistemas acríticos son, por supuesto, mucho más flexibles en cuanto a restricciones que los sistemas críticos. Estos sistemas no requirieren de ningún tipo de planificación previa o garantía de funcionamiento en los todos los casos. Normalmente los sistemas de este tipo son sistemas que buscan obtener una cierta calidad de servicio, como los sistemas multimedia o sistemas que se basan en *best effort* o mejor esfuerzo, estos sistemas nunca garantizan que se cumplan los objetivos, solamente que realizarán todo lo posible por ello, uno de los sistemas de este tipo más conocido es Internet.

Los sistemas basados en la calidad de servicio son muy habituales, uno de los más conocidos son los sistemas multimedia como por ejemplo el *streaming* de video. Este sistema tiene que cumplir ciertos plazos de tiempo para asegurar que las imágenes de video se envían a tiempo para ser reproducidas, sin embargo si una imagen no llega a tiempo para ser reproducida se descarta y se continúa la reproducción en el punto en que estuviese. La calidad del servicio, por tanto, dependerá de la cantidad de imágenes que se puedan enviar al reproductor a tiempo.

Por otro lado los sistemas que aplican técnicas de mejor esfuerzo, como Internet, funcionan de forma que no aseguran en ningún caso que las acciones se lleguen a completar. Estos sistemas habitualmente están dimensionados para un cierto uso, si ese uso se sobrepasa o si alguno de los nodos falla el sistema no garantiza que hayan recursos suficientes o redundantes para que los paquetes lleguen a su destino.

El sistema que se implementará en el presente proyecto queda enmarcado en los sistemas acríticos de calidad de servicio.

2.3 ARQUITECTURAS MULTICORE Y SIMD (SIMPLE INSTRUCTION MULTIPLE DATA)

Desde hace más de medio siglo los problemas que requiere gran cantidad de cálculo se hacen con procesadores. La mayoría de los procesadores actuales siguen la arquitectura conocida como "Arquitectura de Von Neumann" propuesta en 1949 por el científico húngaro John Von Neumann. Esta arquitectura se caracteriza por almacenar en la misma memoria tanto los datos como las instrucciones máquina a ejecutar. Esta fue la primera arquitectura que permitió cambiar la funcionalidad de la máquina sin recablearla físicamente, sino introduciendo las instrucciones a ejecutar en la memoria, naciendo de esta manera el concepto de software.

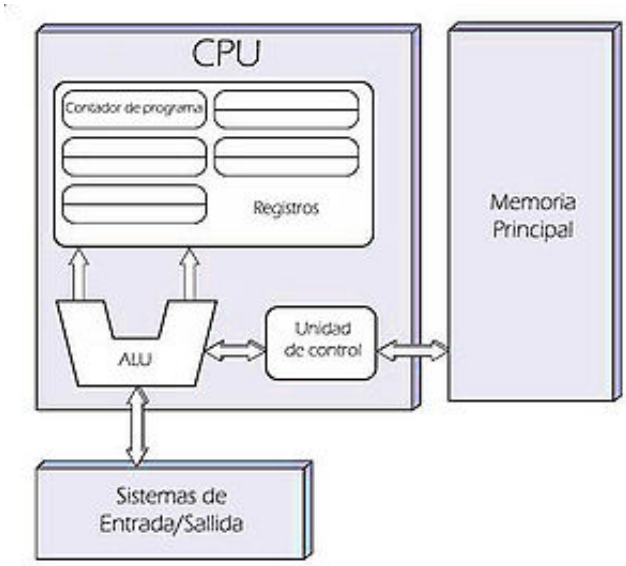

Figura 2.2. Esquema de arquitectura Von Neumann.

Décadas después Intel desarrolló el primer microprocesador de consumo, el Intel 8080 [5], este procesador es la base de gran parte de las arquitecturas actuales.

Durante los años posteriores y hasta el año 2005, la miniaturización de los componentes electrónicos produjo que cada vez hubiese más transistores en los chips, pudiendo así lograr un incremento de frecuencia de reloj en los mismos. El numero de transistores y la frecuencia de trabajo de los procesadores se duplicaba cada 18 meses (tal como indicaba la ley de Moore [5]) dando lugar a los procesadores 80286, 80386, 80486, Pentium® y posteriores.

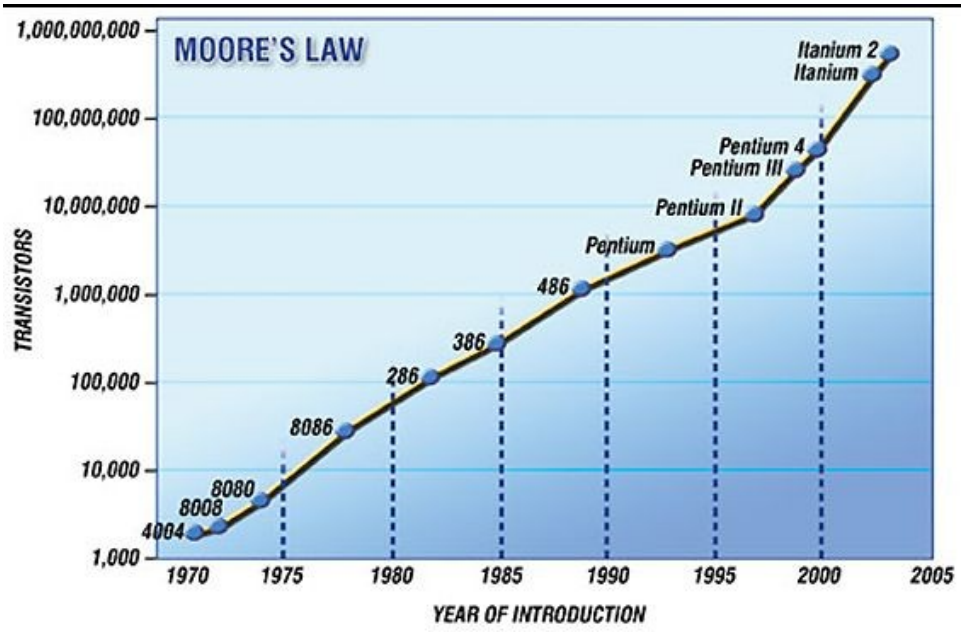

Figura 2.3. Ley de Moore.

Esta reducción progresiva del tamaño del transistor y el aumento implícito de la frecuencia de trabajo proporcionó a la industria del software el *free-food*, es decir, un mismo software funcionaría más rápido cuanto más rápido fuera el hardware sobre el que se ejecutaba debido a que el rendimiento del cálculo computacional era proporcional a la frecuencia de trabajo. No obstante, este *free-food* no podía ser eterno. El aumento de frecuencia comporta un aumento en el consumo y en el calor disipado en un circuito integrado con una relación cúbica respecto a la frecuencia [6].

Debido a este inconveniente los fabricantes comienzan a optar por sistemas que aumentan el rendimiento de cálculo sin aumentar la frecuencia de reloj del procesador. Las soluciones que se implementaron se basaron en la paralelización de operaciones y son principalmente:

- Introducción de operaciones vectoriales, conocidas como SIMD (Simple Instruction Multiple Data). Este tipo de instrucciones permiten ejecutar la misma operación sobre varios datos al mismo tiempo.
- Replicación de núcleos completos de proceso en un mismo chip. Lo que habitualmente se conoce como arquitectura multicore. Son chips cuyo interior alberga dos o más procesadores completos. En la actualidad podemos encontrar procesadores multicore de hasta 8 cores para uso doméstico y hasta

64 para uso profesional. Con la aparición de los multicore se ha conseguido aumentar la potencia de cálculo rápidamente sin aumentar la frecuencia o incluso disminuyendo esta.

2.3.1 INSTRUCCIONES SIMD (SIMPLE INSTRUCTION MULTIPLE DATA)

Los primeros procesadores implementaban instrucciones SISD (Simple Instruction Simple Data). Estas instrucciones solo pueden operar con un dato al mismo tiempo. Son instrucciones escalares. Debido al problema que suponía el incremento de calor disipado por los procesadores al incrementar su frecuencia, se empezaron a implementar, además de instrucciones SISD, instrucciones SIMD (Simple Instruction Multiple Data). Estas instrucciones introdujeron el concepto de paralelismo a nivel de datos, ya que son capaces de operar con varios datos a la vez en una sola instrucción.

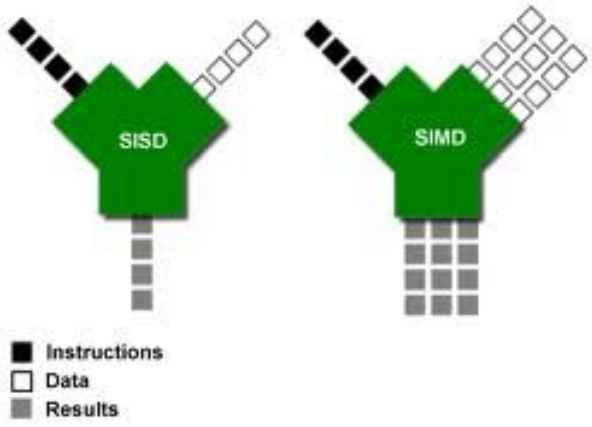

Figura 2.4. Esquema de funcionamiento de instrucciones SISD y SIMD.

La Figura 2.4 muestra conceptualmente la diferencia entre las instrucciones SISD y las instrucciones SIMD. En la parte derecha de la figura se muestra el funcionamiento de las instrucciones SISD, en las cuales cada instrucción opera con un solo dato y proporciona un solo resultado. En la parte derecha se muestran las operaciones SIMD, en las que cada instrucción opera con varios datos y produce tantos resultados a la vez como datos sea capaz de manejar.

Actualmente cada fabricante define su propio conjunto de instrucciones SIMD para sus procesadores, lo que imposibilita su utilización a nivel práctico si se quiere producir código portable. En este proyecto se utilizarán las instrucciones SIMD de las arquitectura de Intel

x64/ia64 a través de la biblioteca ArBB (Array Buildings Block) la cual se detallará en el punto 2.6 de este documento.

Las instrucciones SIMD de la arquitectura Intel x64/ia64 se conocen como SSE (Streaming SIMD Extensions) y AVX (Advance Vector eXtensions). Las instrucciones SSE son las predecesoras de las AVX y tienen un ancho de 128 bits. Las instrucciones AVX se implementan en los últimos modelos de procesadores de Intel, sus características más relevantes son las siguientes[7]:

- Longitud de 256 bits. En versiones futuras se espera ampliarlas a 512 y 1024 bits.
- Soporte para 3 operandos. En versiones anteriores (SSE) sólo se soportaban 2 operandos, siendo uno de ellos sobrescrito con el resultado de la operación.
- Algunas instrucciones toman operandos de 4 registros, haciendo el código más pequeño y rápido al eliminar instrucciones innecesarias, por ejemplo, existen instrucciones que realizan un producto entre dos conjuntos de operandos y posteriormente la suma con un tercero almacenando el resultado en un cuarto operando, o lo que es lo mismo A = B * C + D.
- Se incorporan ciertas operaciones específicas para algoritmos criptográficos como AES (Advanced Encryption System).
- Las instrucciones AVX sólo admiten tipos de coma flotante que cumplan con el estándar IEEE-754.

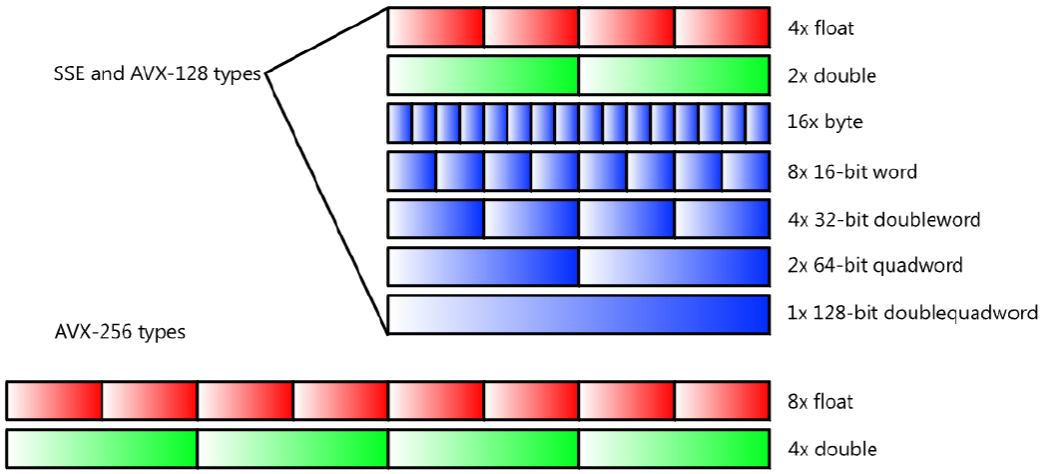

Figura 2.5. Tipos de datos utilizados en instrucciones SSE y AVX de Intel. Fuente: [7].

El gran problema de esta tecnología es la dificultad de su utilización dentro del software de cálculo. La mayoría de los compiladores más usados, como por ejemplo el de Microsoft, Intel o GNU aseguran su utilización, pero se ha demostrado reiteradamente que el uso óptimo de las características que ofrece SSE solo es alcanzable mediante llamadas a código ensamblador (ASM) el cual nunca es portable. En el campo que nos atañe, el de la computación de alto rendimiento, las SSE y AVX (así como otros juegos de instrucciones SIMD en otras plataformas) son usadas en todas aquellas aplicaciones específicas que no busquen la portabilidad. Así pues, bibliotecas numéricas como BLAS o LAPACK no usan SSE o AVX en su versión más portable pero sí que las implementan en versiones más específicas como por ejemplo BLAS de ACML (AMD) y MKL (Intel) totalmente dependientes de la arquitectura.[6]

2.3.2 ARQUITECTURAS MULTIHILO

Las arquitecturas multihilo son aquellas que permiten ejecutar varios hilos ligeros (habitualmente 2) en un solo procesador.

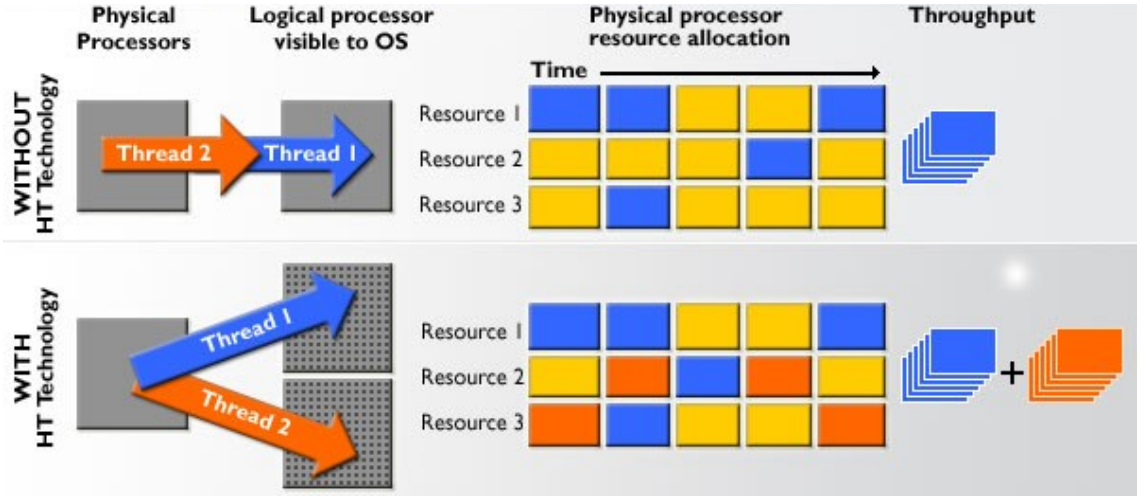

Figura 2.6. Esquema de funcionamiento de tecnología multihilo.

Los procesadores que implementan esta tecnología simulan 2 procesadores lógicos dentro del mismo procesador físico. Sin embargo, de cara a los programas es como si existieran dos procesadores físicos. Para conseguir este efecto no es necesario que se dupliquen todos los componentes del procesador solo algunos de ellos. El resultado es que un mismo proceso pesado puede ejecutar de forma paralela varios hilos aumentando el rendimiento ya que aprovecha mejor el tiempo de procesador. Es importante resaltar que un

procesador multihilo puede ejecutar concurrentemente varios hilos ligeros pero nunca varios procesos pesados ya que no contiene el hardware necesario para ello.

2.3.3 ARQUITECTURAS MULTICORE

Una arquitectura multicore es aquella en la que en un solo circuito integrado se tienen dos o más núcleos completos de procesamiento.

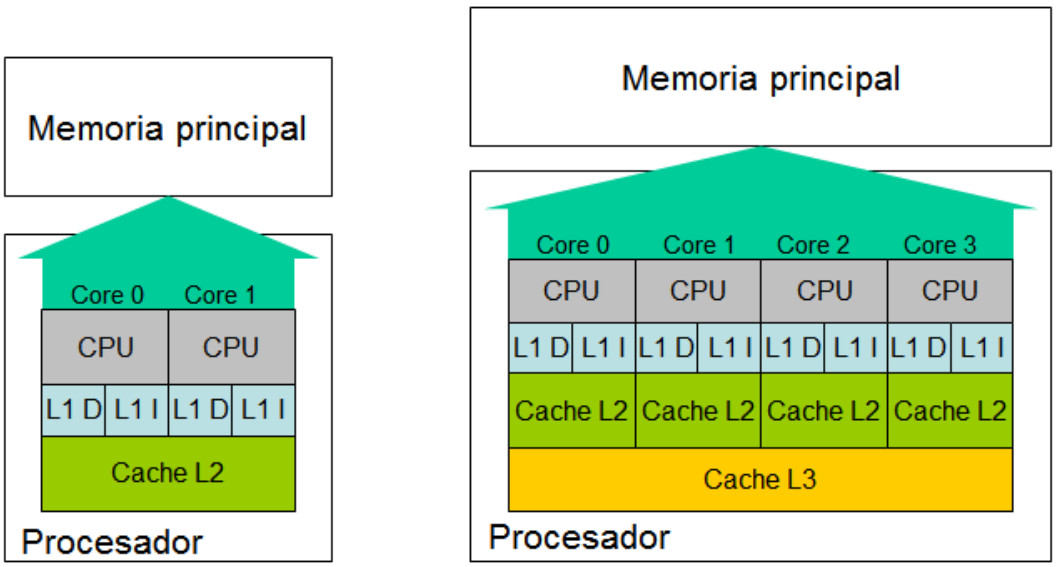

Figura 2.7. Esquema básico de arquitectura multicore para 2 y 4 cores.

Este tipo de hardware permite ejecutar procesos concurrentemente ya que tiene más de una unidad de procesamiento completa. También permite el paralelismo a nivel de aplicación ya que cada núcleo puede ejecutar un proceso ligero o hilo del mismo proceso pesado.

En estas arquitecturas cada núcleo suele tener su propia caché de datos e instrucciones de primer nivel, no es así con los niveles superiores de memoria que suelen estar compartido entre todos los núcleos.

Este tipo de arquitecturas junto con las arquitecturas multihilo han supuesto un gran avance en cuanto a rendimiento se refiere, sin embargo este aumento de rendimiento ya no se produce de forma automática al actualizar a un modelo CPU más moderno con más capacidad de computo, como ocurría en las arquitecturas de un solo core. En estas arquitecturas es necesario que el software se adapte para ejecutar más rápido cuando se aumenta el número

de cores, la responsabilidad del incremento de rendimiento recae ahora también sobre el software que debe estar preparado para ser escalable con el número de cores e hilos de ejecución que proporcionan las arquitecturas multicore.

Conseguir que el software se adapte a esta nueva forma de aumento de rendimiento es ahora responsabilidad de los desarrolladores, los cuales deben ser capaces de implementar software que pueda ejecutarse en varios cores al mismo tiempo. Para ello, los desarrolladores se pueden valer de dos herramientas:

- **Threads nativos:** En la actualidad todos los sistemas operativos tienen la capacidad de manejar varios hilos de ejecución del mismo proceso. Sin embargo cada uno de ellos aborda este problema de forma diferente y proporciona sus propias interfaces haciendo de ellos la mejor opción para plataformas concretas pero anulando por completo la portabilidad del código. Otro de los problemas derivados del uso de threads nativos es que son de muy bajo nivel lo que produce que su implementación sea bastante complicada y la actualización de códigos secuenciales a paralelos sea bastante costosa para el programador.
- **Bibliotecas de paralelización:** Como bibliotecas de paralelización nos referimos tanto a bibliotecas propiamente dichas como a lenguajes de programación específicamente diseñados para implementar aplicaciones concurrentes. Estas herramientas encapsulan los threads nativos del sistema proporcionando una interfaz sencilla al programador para producir código que ejecute de forma paralela. Las grandes ventajas de esta solución es que ofrece portabilidad entre distintas plataformas y que libera al programador de tener que gestionar los hilos directamente, ya que proporcionan un modelo de programación de alto nivel. Entre estas bibliotecas destacan OpenMP e Intel Threading Building Blocks (TBB), ambas serán utilizadas en este proyecto.

2.3.4 MICROARQUITECTURA INTEL SANDY BRIDGE

La arquitectura que se utilizará en este proyecto para probar las implementaciones realizadas será la arquitectura de Intel x64/ia64 en su versión conocida como Sandy Bridge.

El procesador concreto y sus características se describen en los anexos de este documento. En líneas generales la arquitectura Sandy Brigde de Intel es una arquitectura multicore que ofrece chips con 2, 4 y 6 núcleos de procesamiento, también incorpora en algunos de los chips las tecnología Hyperthreading que permite ejecutar dos procesos ligeros o hilos en cada núcleo dando lugar a la ejecución de hasta 12 hilos simultáneos.

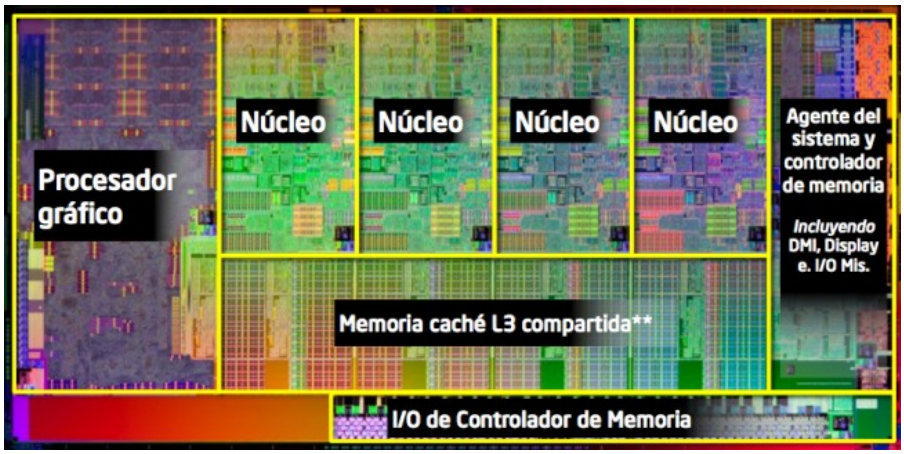

Figura 2.8. Chip Sandy Bridge. Fuente: Intel.

Los chips, contienen además de los propios núcleos de procesamiento 3 niveles de memoria caché. El primer nivel de caché es propio de cada núcleo y se divide en una caché de instrucciones y otra de datos de 32 KB cada una. El segundo nivel también es propio de cada núcleo pero en este caso las instrucciones y los datos se almacenan juntas, tiene una capacidad de 256KB. El último lo compone una caché compartida para todos los núcleos su tamaño varía entre 4 y 12 MB. Dentro del chip también podemos encontrar un procesador gráfico, aunque debido a la naturaleza del proyecto no se ha considerado necesaria la profundización en sus características o funcionamiento.

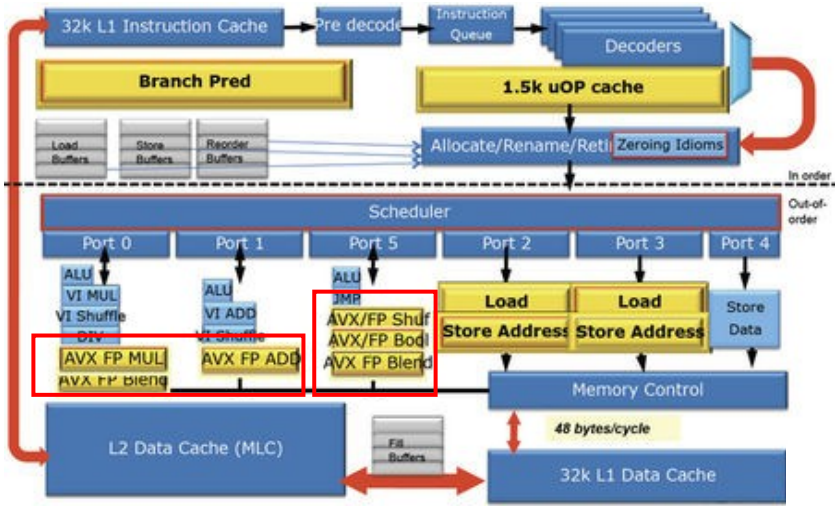

Figura 2.9. Esquema de un núcleo Sandy Bridge.

Una de las particularidades más interesante de la microarquitectura Sandy Bridge es que incorpora el juego de instrucciones AVX. Como se puede observar en la Figura 2.9 los núcleos contienen registros especiales para tratar este tipo de instrucciones.

2.4 OPENMP

OpenMP es una biblioteca de paralelización para programas escritos en C/C++ y Fortran. Su uso se basa en directivas de compilación por lo que es muy portable, ya que no necesita apenas modificación del código de los programas secuenciales para obtener una versión paralela.

2.4.1 MODELO DE EJECUCIÓN

OpenMP utiliza un modelo de ejecución paralela fork/join. Todo programa OpenMP comienza con un único hilo de ejecución que ejecuta de forma secuencial. Este hilo se conoce como hilo principal y ejecuta una región del programa que lo engloba completamente conocida como región principal.

Cuando un hilo, ya sea el principal o cualquier otro alcanza una directiva que define el comienzo de una región paralela crea un grupo de entre 0 y n hilos gestionados por el hilo creador. Para cada hilo creado se genera implícitamente una tarea a partir del código escrito dentro de la región paralela, a partir de este momento cada tarea queda atada al hilo al que le ha sido asignada y ningún otro hilo podrá ejecutarla. Una vez que se ha terminado de ejecutar toda la región paralela por cada uno de los hilos el hilo encargado de gestionar al resto reanuda la ejecución del código secuencial para ese hilo.

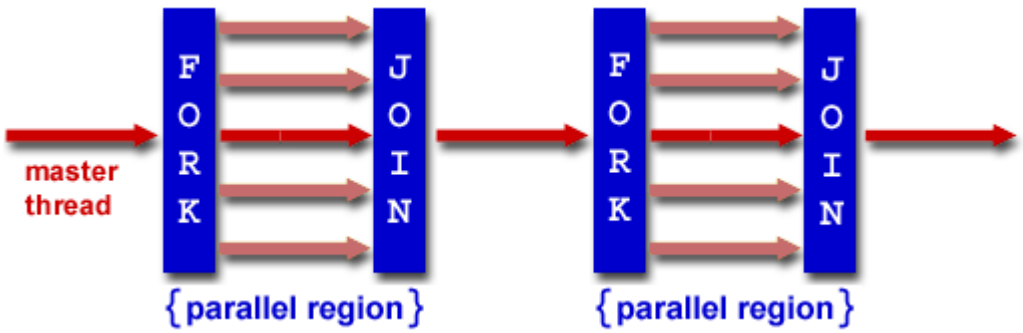

Figura 2.10. Esquema del modelo Fork/Join. Fuente: [19].

OpenMP permite paralelización anidada, es decir cuando dentro de una región paralela se encuentra otra directiva que indica el comienzo de otra región paralela se crean más hilos que ejecuten las tareas de esta región. Una vez terminada la ejecución de la región anidada se continúa con la ejecución de la región de nivel superior.

OpenMP garantiza que todas las tareas que han sido asignadas a hilos en regiones paralelas terminan antes de que el hilo principal reanude su ejecución y/o el programa finalice. Para el resto de casos OpenMP ofrece directivas de sincronización que deben ser utilizadas explícitamente por el programador.

La principal función de OpenMP es la paralelización de códigos regulares, como bucles `for`. Cuando se paraleliza una estructura de este tipo OpenMP se encarga de dar a cada hilo una iteración del bucle, para repartir las iteraciones del bucle entre los hilos utiliza un planificador cuya política puede definir el programador. Existen 3 políticas disponibles:

- Estática (*static*): Con esta política se asignan las iteraciones a cada thread antes de empezar la ejecución de las mismas. Las iteraciones son divididas entre el número de threads disponibles. Se puede indicar el número de iteraciones contiguas que se asignan a cada thread.
- Dinámica (*dynamic*): En esta política a cada thread se le asigna un pequeño número de las iteraciones totales. El programador puede indicar que número de iteraciones se le asignan a cada thread, por defecto es una. Cuando un hilo termina de ejecutar todas las iteraciones que le han sido asignada se le asigna más hasta que se termine la ejecución de todas.
- Guiada (*guided*): Con esta política el usuario define el número mínimo de iteraciones que se asignan a un thread cada vez. Al principio de la ejecución del bucle a cada thread se le asigna un número alto de iteraciones. Cuando un hilo termina se le asignan más iteraciones, pero cada vez menos hasta llegar al valor mínimo definido por el programador.

A partir de su versión 3.0, OpenMP ofrece paralelización centrada en tareas. Este tipo de paralelización se basa en crear tareas y mapearlas a hilos, de forma que se consigue que los hilos estén más tiempo realizando tareas útiles y por tanto un mayor rendimiento. En el desarrollo de este proyecto no se utilizará esta característica ya que OpenMP se utilizará para hacer una paralelización de grano fino, mientras que la paralelización de grano grueso, es decir las tareas se realizará con la biblioteca Intel Threading Building Blocks, la cual se explica en el punto 2.5 de este documento.

2.4.2 MODELO DE MEMORIA

OpenMP ofrece un modelo de consistencia relajada en un paradigma de programación de memoria compartida. Todos los hilos en OpenMP tienen acceso a memoria y se les permite tener su propia vista temporal de la misma. Esta vista temporal de la memoria permite a los hilos tener copias propias de variables en caché y evitar así ir a memoria cada vez que se accede a una variable. Por otra parte cada hilo tiene, además, acceso a una memoria privada que no puede ser accedida por ningún otro hilo.

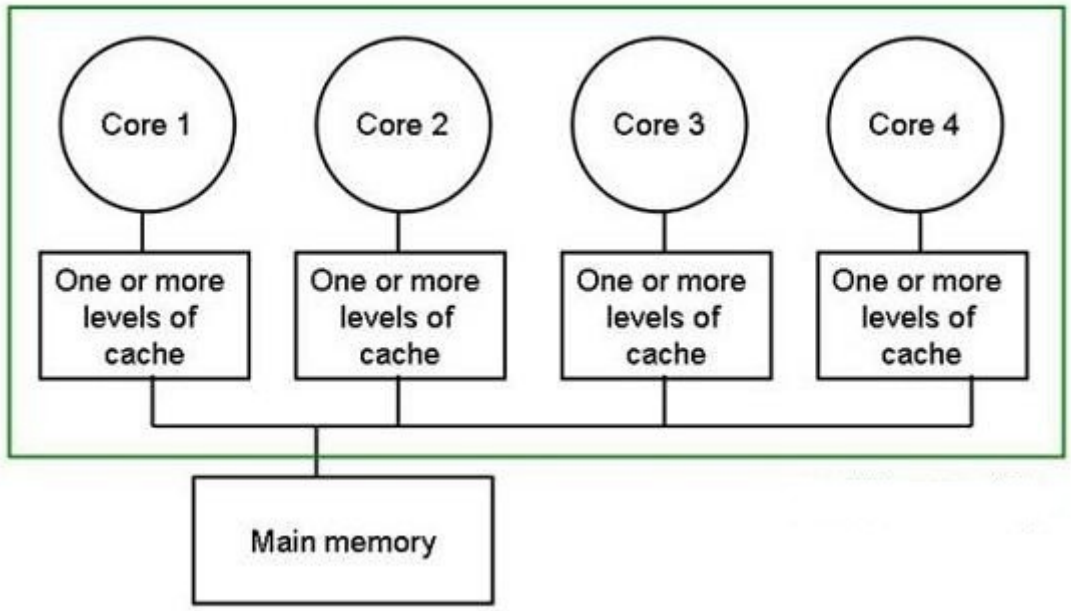

Figura 2.11. Esquema de un sistema de memoria compartida. Fuente: [20].

En OpenMP existen dos tipos de variables, compartidas y privadas. En el primer caso existe solamente una copia de la variable en la memoria, y esta sea accedida por todos los hilos que ejecuten la región donde se encuentra. En el caso de las variables privadas se crea una copia en memoria por cada tarea o hilo que ejecuta la región de código en la que se encuentra la propia variable, esta copia es privada para cada hilo. La creación o acceso a una variable privada por parte de los hilos no modifica la variable original y el resultado final de esta dependerá de la directiva concreta que defina la región paralela.

Algo a tener en cuenta en OpenMP es que las operaciones de memoria no son atómicas debido a que cualquier acceso, ya sea de lectura o escritura, puede implementarse con varias instrucciones máquina por lo que es responsabilidad del programador definir qué operaciones deben realizarse de forma atómica a través del uso de las directivas atómicas que define OpenMP.

Como en cualquier código ejecutado por varios hilos de forma concurrente, en OpenMP se pueden producir condiciones de carrera cuando varios hilos intentan acceder a una variable en modo escritura o cuando uno intenta un acceso de lectura al mismo tiempo que otro realiza una escritura. Para solventar este problema OpenMP incluye directivas de sincronización que permiten al programador definir el orden en que los diferentes hilos acceden a una misma variable compartida, aunque no se debe olvidar que esta cuestión es responsabilidad del programador y que en caso de que se produzcan condiciones de carrera el resultado de las operaciones es indefinido y por lo tanto será distinto para cada ejecución del código.

Como se ha dicho al comienzo de este punto OpenMP utiliza un modelo de consistencia relajada. Esto se debe a que no es necesario que la vista temporal de memoria que tienen los hilos sea consistente con la memoria principal continuamente. Para los casos en que sea necesario tener una visión actualizada de una variable OpenMP permite forzar la consistencia de memoria con la directiva *flush*. Esta directiva se aplica a conjuntos de variables conocidas como *flush-set* y mediante ella se puede forzar la exportación a todos los hilos que ejecutan una región paralela de una variable modificada por otro hilo. Además de esto la directiva *flush* restringe la reordenación de operaciones de acceso a memoria asociadas a la variable o variables sobre las que se aplica la directiva.

Las restricciones que introduce la directiva *flush* respecto a la reordenación de código garantizan que [8]:

- Si la intersección de los *flush-set* de dos *flush* llevados a cabo por dos hilos distintos es no vacía, entonces ambos *flush* deben ser completados en algún orden secuencial visto por todos los hilos.
- Si dos hilos realizan operaciones sobre la misma variable estas deben ser completadas en algún orden secuencial visto por todos los threads
- Si la intersección de los *flush-set* de dos *flush* es vacía, entonces los hilos pueden observar los *flush* en cualquier orden.

En definitiva, la directiva *flush* garantiza la consistencia entre la vista temporal de un hilo y la memoria aunque siempre debe ser el programador el que se encargue de asegurarlo. Por otra parte la directiva *flush* no previene por sí misma las condiciones de carrera que se pueden dar en un código paralelo si no se utiliza junto con directivas atómicas.

2.5 INTEL THREADING BUILDING BLOCKS (TBB)

Intel Threading Building Blocks (TBB) es una biblioteca de plantillas que extiende C++ abstrayendo la gestión de hilos y permitiendo paralelizar algoritmos de forma sencilla. A diferencia de otras bibliotecas, en TBB se especifican tareas en vez de hilos, y es la biblioteca la que se encarga de asignar a cada hilo disponible en el sistema las tareas definidas por el programador de la forma más eficiente posible. Este enfoque permite que los programas escritos en C++ y ejecutados en arquitecturas multicore sean escalables con el número de procesadores.

TBB utiliza el estándar de C++ y no requiere de otros lenguajes ni de compiladores específicos. Esta capacidad de utilizarlo en cualquier sistema con cualquier compilador de C++ hace que sea muy portable y por lo tanto una gran alternativa en la actualidad. TBB utiliza plantillas para los patrones comunes de paralelización, por lo que permite al programador paralelizar código sin tener que preocuparse apenas por la sincronización, el balanceo de carga o la optimización en el acceso a memoria. Los programas paralelizados con TBB pueden ejecutar en máquinas con un solo procesador o con varios. Además permite paralelismo anidado, por lo que se pueden implementar componentes complejos a partir de componentes más básicos de forma sencilla.

Como ya se ha indicado TBB se basa en la definición de tareas por parte del programador que después serán asignadas de forma automática a los hilos del sistema, esto permite especificar el paralelismo de forma más cercana al programador, abstrayéndose de la cercanía de los hilos con el hardware y obtener un resultado mejor que con hilos nativos.

El objetivo del programador cuando implementa algoritmos paralelos es la escalabilidad. El modelo de tareas ofrecido por TBB permite aprovechar al máximo el número de hilos disponibles en un sistema, obteniendo así una gran escalabilidad. TBB se diferencia de otras bibliotecas de paralelización en varias cosas:

- **Especifica tareas en vez de hilos:** La mayoría de las bibliotecas de programación paralela requieren que el programador cree y gestiones los hilos, esto además de ser tedioso puede ser muy ineficiente ya que los hilos son elementos de muy bajo nivel, muy pesados de construir. TBB en cambio trabaja con tareas, las cuales son mucho menos pesadas de construir que los hilos y fáciles de planificar, por lo que se obtiene un uso más eficiente del sistema. Por otra parte al evitar la

programación directa con hilos nativos se obtienen programas más portables, fáciles de programar y un código fuente más legible.

- **Pensado para conseguir rendimiento:** La mayoría de las bibliotecas de paralelización soportan diferentes tipos paralelismo con hilos, pero en general las bibliotecas de propósito general tienden a ser herramientas de bajo nivel que ofrecen una forma de hacer las cosas pero no una solución. Sin embargo TBB se centra en la paralelización de tareas computacionalmente intensivas, desarrollando soluciones de alto nivel.
- **Compatibilidad con otras bibliotecas de programación:** TBB puede coexistir en el mismo programa con otras soluciones de paralelización. Esto es muy importante ya que no fuerza al programador a tener que escoger una sola tecnología para todo el programa. Con TBB es posible utilizar cualquier otra herramienta de paralelización como por ejemplo OpenMP o hilos nativos. Además esta característica permite que podamos utilizar bibliotecas creadas por terceros aunque estén creadas con otras tecnologías de paralelización.
- **Generación de programas altamente escalables:** Lo habitual al programar de forma concurrente es separar el programa en bloques funcionales y asignar a cada uno de ellos un hilo separado. Esta solución no suele ser escalable ya que el número de bloques funcionales suele ser fijo y por lo tanto al ejecutar el programa en un sistema cuyo número de hilos es mayor que el número de bloques definidos en el programa muchos de ellos se desaprovechan. En contraste a esto TBB utiliza paralelismo de datos haciendo que los hilos trabajen de forma más eficiente al definir tareas que pueden ser asignadas a hilos dinámicamente en tiempo de ejecución, aprovechando de esta forma el mayor número de hilos de ejecución disponibles en el sistema.
- **Programación genérica:** Las bibliotecas tradicionales definen interfaces en términos de clases específicas o clases base. TBB utiliza programación genérica basada en plantillas de C++. La esencia de la programación genérica es escribir lo mejor posible algoritmos con las mínimas restricciones. La biblioteca de plantillas estándar (STL) de C++ es un buen ejemplo donde las interfaces se definen por requisitos en los tipos. La programación genérica permite personalizar los componentes a cada necesidad específica.

TBB permite diferentes formas de paralelismo según las necesidades específicas de cada aplicación. Las diferentes formas no son excluyentes entre ellas por lo que se pueden combinar para conseguir la mayor eficiencia y escalabilidad posible.

2.5.1 PARALELISMO DE DATOS

El paralelismo de datos se basa en tomar una gran cantidad de datos y aplicar la misma operación sobre cada uno de ellos de forma concurrente. La Figura 2.12 muestra un ejemplo gráfico de ello, cada letra es convertida en mayúscula concurrentemente aplicando la misma operación a un conjunto de datos.

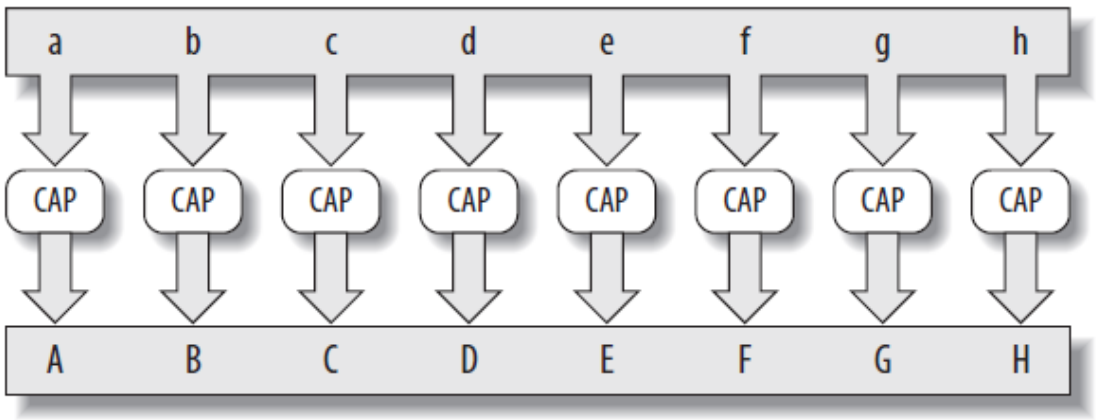

Figura 2.12 Paralelismo de datos. Fuente: [9].

2.5.2 PARALELISMO DE TAREAS

El paralelismo de datos está limitado por la cantidad de datos que se quieren procesar. Otra forma de afrontar el paralelismo es el modelo de paralelismo de tareas. El paralelismo de tareas se basa en realizar diferentes tareas sobre el mismo conjunto de datos. La ventaja principal de este modelo de programación es que permite aprovechar al máximo el tiempo de procesador del sistema ya que cuando una tarea no puede ser ejecutada, ya sea porque se ha bloqueado en un mutex o porque en ese momento no es necesario ejecutarla se puede dedicar ese hilo de ejecución a otra tarea, manteniendo de esta manera el procesador ocupado el máximo tiempo posible.

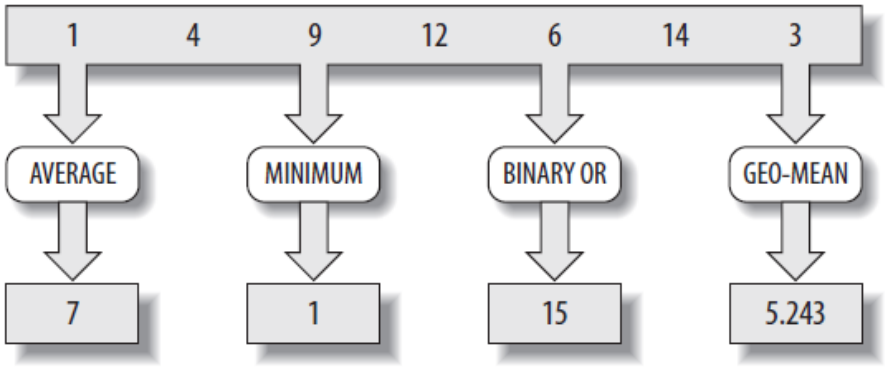

Figura 2.13. Paralelismo de tareas. Fuente: [9].

La Figura 2.13 muestra gráficamente como se aplican diferentes operaciones matemáticas sobre el mismo conjunto de datos de forma paralela. En este caso se calculan diferentes valores tomando como datos de entrada un único vector compartido en todas las tareas.

2.5.3 *PIPELINING* (PARALELISMO DE DATOS Y TAREAS)

Este tipo de paralelismo se basa en hacer pasar los datos por un flujo de tareas, como si de una tubería se tratase. Cada dato es procesado en varias etapas, mientras avanza por el *pipeline*. Con esta técnica los datos son procesados de forma más rápida y eficiente ya que diferentes datos pueden estar en diferentes etapas al mismo tiempo como muestra la Figura 2.14.

Figura 2.14 *Pipeline*. Fuente: [9].

Los *pipelines* pueden ser más complejos, permitiendo saltar etapas para ciertos datos o haciendo que en determinadas etapas se puedan tratar varios objetos en paralelo.

Hay que tener en cuenta que TBB permite tanto la mezcla entre sí de todos los modelos de programación paralela que implementa como la mezcla con otros modelos como pueden ser OpenMP o hilos nativos, por lo que siempre se puede optar por una solución híbrida, que habitualmente será la que mejor rendimiento ofrezca.

2.5.4 PLANIFICACIÓN DE TAREAS

TBB implementa un planificador de tareas muy eficiente. Cuando TBB planifica las tareas intenta utilizar todos los cores disponibles en el sistema y a ser posible explotando la localidad de los datos. Por esta razón en ocasiones unos cores tienen más carga de trabajo que otros. Estas tareas se organizan en colas en cada core a la espera de ser ejecutadas. Para balancear la carga y no mantener cores ociosos mientras otros tienen tareas encoladas en espera TBB implementa un sistema de robo de tareas o *task stealing* capaz de detectar cuando un core no tiene tareas que ejecutar y está ocioso y pasar tareas de otros cores más cargados de trabajo a éste, aprovechando de esta forma los recursos del sistema al máximo.

2.5.5 GESTIÓN DE MEMORIA

Uno de los problemas a la hora de desarrollar aplicaciones paralelas es cómo éstas acceden a memoria. Las aplicaciones paralelas están compuestas por varios hilos que acceden a la misma memoria (en paradigmas de memoria compartida) por lo que puede suceder que en ciertos momentos accedan a las mismas posiciones de memoria a la vez, modificándolas y dejándolas en un estado indefinido, lo cual siempre es un comportamiento no deseado. Este fenómeno se conoce como *condiciones de carrera*.

Para evitar este problema TBB pone a disposición del programador varias herramientas capaces de controlar el acceso a memoria y evitar que se produzcan condiciones de carrera.

2.5.5.1 EXCLUSIÓN MUTUA

Para evitar que varios hilos ejecuten una zona de código que se considera crítica (porque en ellas se utilizan recursos compartidos) se utilizan los llamados mutex. Un mutex es un objeto que evita que varios hilos ejecuten el mismo fragmento de código. Para conseguir esto, cuando hilo pasa un mutex antes de entrar en una zona crítica el mutex comprueba que ninguno otro hilo haya pasado antes. Si esto es así, se dice que el hilo ha adquirido un cerrojo y el mutex no deja pasar más hilos. Cuando el hilo que se encuentra en la zona crítica termine liberará el cerrojo permitiendo que otro hilo lo adquiera y ejecute la zona crítica. Mientras tanto el resto de hilos que quieran ejecutar la zona crítica quedarán bloqueados en el mutex.

TBB ofrece un amplio repertorio de mutex[10]:

Recursivos (*recursive_mutex*): Este tipo de mutex permite a un hilo obtener un cerrojo en varias ocasiones de forma anidada. Otro hilo no podrá acceder al mutex hasta que el hilo que ha adquirido los cerrojos libere todos ellos.

Sin bloqueo (*spin_mutex*): Este tipo de mutex funcionan como un mutex normal, exceptuando que los hilos que intentan acceder cuando otro ha adquirido el cerrojo no se bloquean sino que realizan una espera activa. Esto produce que no se haga un cambio de contexto de hilo en el procesador, siendo por lo tanto una operación menos pesada que en el caso de los mutex normales. Sin embargo, solo deben utilizarse cuando la zona crítica es muy pequeña ya que sino, el mantener un núcleo ocupado en una espera activa durante mucho tiempo puede disminuir el rendimiento del sistema. Otro problema que presenta este tipo de mutex es su uso en procesadores multihilo, ya que al no tener estos todos sus componentes por duplicado la espera activa en un componente único puede producir que otro hilo que ejecuta en el mismo procesador quede también bloqueado.

Con cola de peticiones (*queuing_mutex):* Utiliza la misma técnica de espera que los mutex sin bloqueo pero en este caso se garantiza que lo hilos que permanecen esperando la liberación del cerrojo lo adquirirán en el orden exacto en que lo ha solicitado.

TBB también implementa un tipo diferente de mutex llamado *Reader Writer Mutex.* Este mutex está especialmente pensado para el problema del productor/consumidor, en que existen varios productores y varios consumidores. El funcionamiento de este mutex se basa en que solo se bloquea si el hilo que adquiere el cerrojo va a modificar una variable compartida, es decir, permite que varios hilos accedan a la misma zona de memoria siempre que este acceso sea de lectura. Para ello el programador debe indicar qué tipo de acceso hará el hilo que intenta obtener el cerrojo. Al igual que en el caso de los mutex normales existen varias versiones de este tipo de mutex.

Sin bloqueo (*spin_rw_mutex*): Este tipo de mutex es como el mutex normal sin bloqueo pero con la capacidad de distinguir entre procesos lectores y escritores.

Con cola de peticiones (*queuing_rw_mutex):* Este tipo de mutex es como el mutex normal con cola de peticiones pero con la capacidad de distinguir entre procesos lectores y escritores.

2.5.5.2 TIPOS ATÓMICOS

Los tipos atómicos son tipos que garantizan que su acceso se realiza de forma atómica, es decir, sin que haya interrupciones para las siguientes instrucciones[10]:

- Lectura: La lectura atómica garantiza que ningún otro proceso del sistema cambiará el valor de la variable mientras ésta se está leyendo.
- Escritura: La escritura atómica garantiza que sólo un hilo modificará la variable al mismo tiempo.
- Captación y suma (*fetch_and_add(<T>)*): Esta función recibe un valor como parámetro que será sumado de forma atómica al tipo atómico sobre el que se invoque la función.
- Captación e incremento (*fetch_and_increment()*): Suma 1 al valor del tipo atómico sobre el que se invoca la función.
- Captación y almacenamiento (*fetch_and_store(<T>)*): Devuelva el valor almacenado en el tipo atómico y posteriormente escribe en el tipo atómico el valor que recibe como parámetro.
- Comparación e intercambio (*compare_and_swap(<T>, <T>*): Recibe dos parámetros. El tipo atómico tomará el valor del primer parámetro si su valor anterior coincide con el valor del segundo parámetro.

En el caso de que el tipo atómico sea un entero o un puntero también son atómicas las siguientes operaciones[10]:

- Captación y decremento (*fetch_and_decrement()*): Resta uno al valor del tipo atómico sobre el que invoca la función.
- Operador postincremento (++): evaluación e incremento atómico de la variable que actúa como operando.
- Operador postdecremento (--): evaluación y decremento atómico de la variable que actúa como operando.
- Operador suma y asignación (+=): Suma de un valor a valor del tipo atómico y asignación del resultado al tipo atómico.
- Operador resta y asignación (+=): Resta de un valor a valor del tipo atómico y asignación del resultado al tipo atómico.

Una de las herramientas más útiles de C++ es la *Standard Template Library* (STL). En esta biblioteca se encuentran, entre otras cosas, los contenedores estándar de C++. Estos contenedores son estructuras de datos capaces de almacenar cualquier tipo, ya sea nativo del lenguaje o definido por el usuario. Además es posible aplicar a cada uno de estos contenedores cualquiera de los algoritmos que define la propia biblioteca y que también están definidos como plantillas. TBB define sus propias plantillas de contenedores, las cuales se pueden utilizar como si de contenedores de la biblioteca estándar de C++ se tratasen pero con el añadido de que además son concurrentes. Es decir, se puede trabajar con ellas en programas paralelos sin tener que tener en cuenta los problemas de condiciones de carrera que pueden surgir del acceso de varios hilo a uno de los contenedores. Los contenedores que define TBB son los siguientes[10]:

- *Concurrent vector*: Es una estructura de datos lineal en la que se puede insertar en cualquier posición y leer de cualquier posición.
- *Concurrent queue*: Es una estructura de datos lineal tipo FIFO, en la que se inserta por el final y se extrae por el principio.
- *Concurrent priority queue*: Igual que en el caso anterior pero con prioridad, es decir, los elementos que se definan con más prioridad se colocarán más cerca de la cabeza.

2.6 INTEL ARRAY BUILDING BLOCKS (ARBB)

ArBB es un *framework* de paralelización en fase de pruebas desarrollado por Intel. Está diseñado para proporcionar paralelismo a varios niveles, nivel de tarea, nivel de dato y a nivel de instrucción. En el modelo de programación de ArBB el sincronismo está implícito, asegurando la consistencia de los resultados sin intervención del programador. El modelo de memoria ofrece aislamiento y es capaz de soportar tanto esquemas de memoria distribuida como de memoria compartida.

ArBB soporta un modelo de programación determinístico y estructurado. Es un lenguaje embebido cuya sintaxis es implementada como un API. Sin embargo en la práctica es utilizado como una biblioteca.

Los tipos de ArBB se definen utilizando mecanismos del estándar de C++. Los tipos de ArBB son tanto valores enteros como de coma flotante y colecciones de estos. Las operaciones sobre los tipos definidos por ArBB se especifican por el programador de la forma habitual. Sin embargo estas secuencias pueden ser capturadas por ArBB y transformadas dinámicamente (en tiempo de ejecución) en instrucciones máquina vectoriales aptas para un gran número de procesadores. ArBB utiliza una semántica de *paso por valor* en las asignaciones. Esto simplifica bastante la tarea del programador ya que las expresiones sobre colecciones actúan como si una nueva colección fuese creada y las asignaciones como si se copiase cada elemento de la colección.

2.6.1 MAQUINA VIRTUAL DE ARBB

El API de ArBB se sitúa en la parte superior de una máquina virtual. Esta máquina virtual se encuentra aislada en una biblioteca y se encarga de la compilación y la paralelización.

La máquina virtual es la encargada de mapear el paralelismo abstracto que se define en el código en los mecanismos que ofrece la máquina donde se ejecuta el código. El ancho de las instrucciones vectoriales y el número de threads disponibles en la máquina quedan ocultos por la máquina virtual. Debido a esto y como las instrucciones vectoriales son generadas dinámicamente en tiempo de ejecución el código escrito con ArBB es portable a través de numerosos conjuntos de instrucciones vectoriales (SSEx, AVX, MIC) incluso sin tener que recompilar el código. El hecho de autogestionar automáticamente los hilos del sistema hace que los programas escalen automáticamente con el número de procesadores.

2.6.2 LENGUAJE DE ARBB

El lenguaje de ArBB está diseñado para mejorar la productividad de dos formas. En primer lugar presenta un modelo simplificado de programación que se centra en el paralelismo, la localidad en memoria y en patrones comunes de computación. Esto simplifica la tarea del programador sobre todo debido a que los patrones deterministas usados reducen los errores comunes de programación paralela como las condiciones de carrera o los interbloqueos. El código generado con ArBB es muy corto y muy cercano a la especificación matemática del problema. En segundo lugar, el código es portable, esto incrementa la productividad ya que no es necesaria ninguna operación adicional para mover aplicaciones a nuevas máquinas, incluso con procesadores totalmente distintos arquitecturalmente.

2.6.3 COMPILACIÓN Y EJECUCIÓN DE PROGRAMAS CON ARBB

La arquitectura de compilación de ArBB se muestra en la Figura 2.15. El *back end* del compilador consiste en un optimizador de alto nivel (HLO), un optimizador de bajo nivel (LLO) y un generador de código que genera código optimizado para la plataforma de ejecución (CCG). Para el manejo de hilos se utiliza la biblioteca de *threading* (TRT), esta biblioteca está basada en Intel Threading Building Blocks (TBB) y comparte un gestor de recursos con otras herramientas de paralelización, ya sean de Intel o de terceros. La *Heterogeneus Runtime (HRT)* se utiliza para lograr un balanceo de carga.

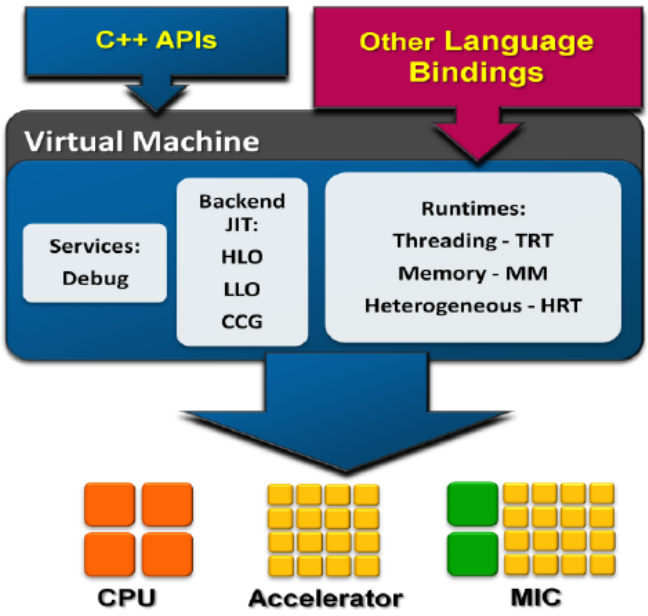

Figura 2.15. Arquitectura de compilación de código ArBB.

En la Figura 2.16 se muestra un esquema del funcionamiento de ArBB. El compilador de C++ utiliza las cabeceras y las plantillas de ArBB para compilar las funciones embebidas del lenguaje como *call* o *bind* cuya implementación se encuentra en la biblioteca dinámica de ArBB. Cuando estas funciones son llamadas, la biblioteca de ArBB prosigue con el control de flujo especificado por el código C++ e interpreta sólo el núcleo de ArBB para construir una representación intermedia con el contexto. Posteriormente en tiempo de ejecución la representación intermedia generada antes es recogida por el compilador *JIT* (*just-in-time*). Este compilador es habitualmente lanzado por la ejecución de la función *call()*. La llamada a la función *call()* produce las siguientes acciones:

- Las sentencias ArBB de C++ son ejecutadas y guardas como una representación intermedia.
- El compilador *JIT* es lanzado para generar código optimizado para la plataforma de ejecución.
- El código generado es ejecutado.

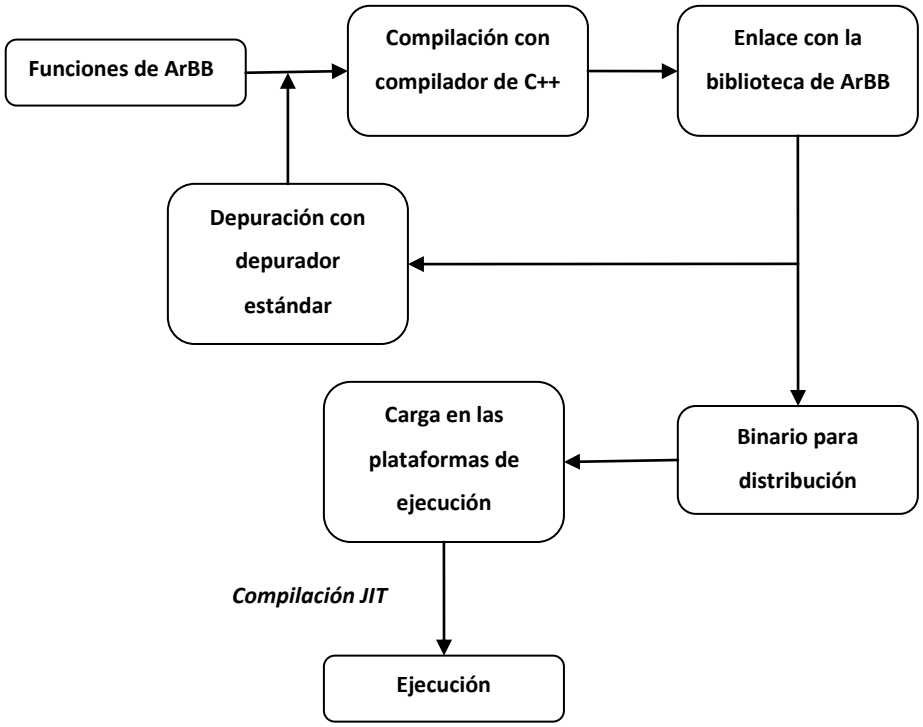

Figura 2.16. Ciclo de vida de ArBB

2.6.4 OPTIMIZACIONES DE CÓDIGO

El compilador de ArBB optimiza la representación intermedia que realiza la biblioteca para mejorar el rendimiento a diferentes niveles. El optimizador de alto nivel (HLO) lleva a cabo una optimización de código independiente de la arquitectura para reducir el consumo de memoria y por tanto su acceso, la sobrecarga producida por la creación de hilos y mejorar la localidad de los datos. El optimizador de bajo nivel (LLO) toma el código generado por el HLO y genera código en instrucciones vectoriales para la plataforma en que se ejecute y lo paraleliza mediante threads. Finalmente el generador de código (CCG) genera un binario optimizado para ejecutar en la plataforma destino.

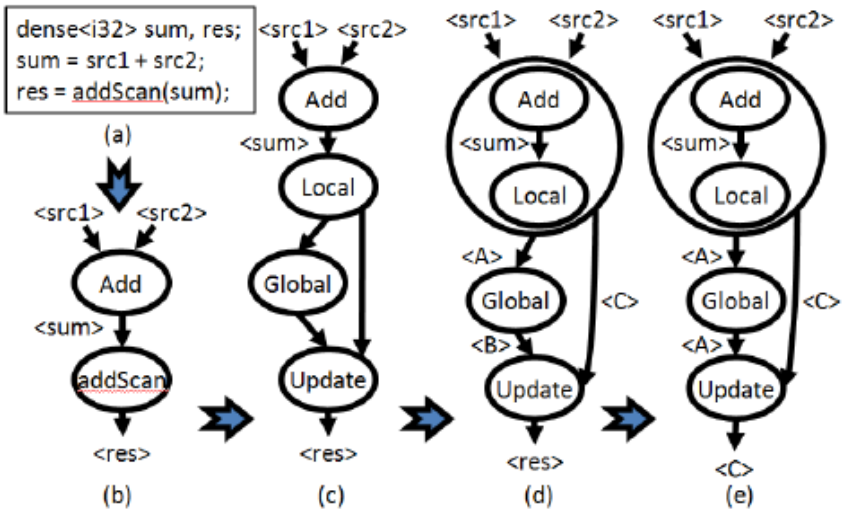

Figura 2.17. Transformaciones del HLO.

En la Figura 2.17 se representan las diferentes etapas de optimización de código en el HLO. La primera fase lleva a cabo una transformación del código a un mapa de funciones y una descomposición en sub-funciones al igual que las optimizaciones clásicas de los compiladores. Estas trasformaciones se llevan a cabo repetidamente. Posteriormente partiendo del mapa de funciones generado se transforman las operaciones escalares en operaciones vectoriales para conseguir paralelismo de datos. La descomposición en sub-funciones divide la función original en tres partes, llamadas sub-primitivas, local, global y actualización (*update).* Las sub-primitivas locales ejecutan operaciones a nivel de elemento en threads. La sub-primitiva global recoge los resultados de las sub-primitivas locales. La sub-primitiva *update* actualiza el resultado final en memoria de cada tarea usando el resultado de la sub-primitiva global y si es necesario de las locales. En las siguientes dos etapas se fusionan operaciones para aumentar el rendimiento.

Tras las optimizaciones realizadas por el HLO, el LLO lleva a cabo la paralelización tanto a nivel de datos como de tarea. La paralelización a nivel de tarea sólo se realiza en los bucles `for` que no tienen dependencias entre iteraciones. El propio LLO es el que ajusta la política de partición de datos. Respecto a la vectorización de operaciones, ArBB se encarga del balanceo de carga, de la alineación de los datos en memoria y también implementa algoritmos SIMD eficientes.

2.7 OTRAS SOLUCIONES DE PARALELIZACIÓN

Además de las soluciones de paralelización descritas y que serán las que se utilicen en el proyecto existen otras soluciones en el mercado tanto propietarias como estándares que también son ampliamente utilizadas.

Uno de los componentes más utilizados para llevar a cabo computación paralela, además del procesador, son las GPU (*Graphic Processor Unit*). Las tarjetas gráficas se llevan utilizando desde los años 80 para liberar a la CPU de realizar los cálculos necesarios para el dibujado de primitivas gráficas. En la actualidad se ha ampliado su uso para realizar cómputo de propósito general debido a que son capaces de ejecutar cientos e incluso miles de hilos de forma concurrente. Esto hace que hayan cobrado gran importancia en entornos HPC (*High Performance Computing*) y que cada vez se estén extendiendo con más fuera de estos entornos.

2.7.1 CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

CUDA (*Compute Unified Device Architecture*) es un conjunto de herramientas desarrollado por nVidia que permite a los desarrolladores utilizar una variación del lenguaje C/C++ para codificar algoritmos en tarjetas gráficas de nVidia. Además de C/C++ se pueden utilizar otros lenguajes como Java, Python o Fortran a través de *wrappers.*

Los programas CUDA utilizan tanto la CPU de la máquina, conocida como *host*, como las GPU's disponibles, conocidas como *devices.* CUDA utiliza una extensión de C con identificadores adicionales para especificar si las funciones son definidas para ejecutar en la CPU, en la GPU o en ambas. También introduce identificadores para especificar en qué memoria se almacenan las variables.

El código se escribe habitualmente en ANSI C/C++, aunque se pueden utilizar otros lenguajes) y se compila con el compilador para C/C++ de nVidia (nvcc). Como resultado de esto se obtienen dos tipos de ficheros, los ficheros fuente para el host y los ficheros con código PTX, el cual es un pseudo-ensamblador para la GPU. Los ficheros fuente para el host se compilan con cualquier compilador de C/C++ estándar. Posteriormente, en tiempo de ejecución, el driver CUDA se encarga de compilar el código PTX para el hardware concreto de la máquina. Esto se hace mediante compilación JIT (*Just In Time*) y permite realizar código portable entre

máquinas con diferente hardware CUDA. En la Figura 2.18 se muestra el esquema de compilación de CUDA.

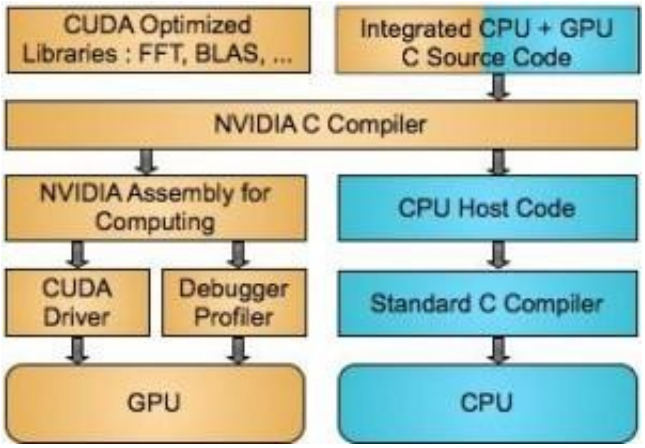

Figura 2.18. Esquema de compilación en CUDA

Las GPU's CUDA están compuestas de varios *Streaming Multiprocessors* (SM). Cada SM es comparable a una CPU con una interfaz para captación de instrucciones, decodificación, unidades funcionales y registros. Cada SM tiene varias unidades funcionales, llamadas cores, los cuales ejecutan diferentes threads con la misma instrucción sobre diferentes datos, por lo que son unidades SIMD según la taxonomía de Flynn.

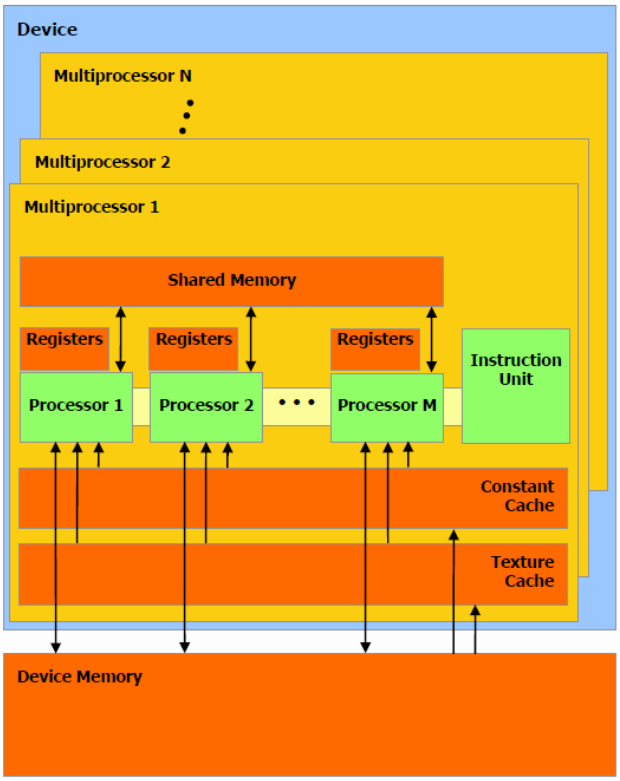

Figura 2.19. Arquitectura CUDA. Fuente: [11].

La unidad básica de operación en CUDA es el thread, en cada dispositivo CUDA se pueden ejecutar a la vez miles de threads. Dentro del dispositivo los threads se organizan en bloques (*blocks*) y estos a su vez en mallas (*grids*). Cada *grid* de threads solo puede ejecutar una función (*kernel*) al mismo tiempo.

La memoria también se organiza jerárquicamente. Cada hilo tienen su propia memoria privada. A su vez por cada bloque de hilos existe una memoria compartida por todos ellos. Finalmente se tiene una memoria global compartida por todos los hilos del sistema. En la Figura 2.20 se muestra un esquema de ambas jerarquías.

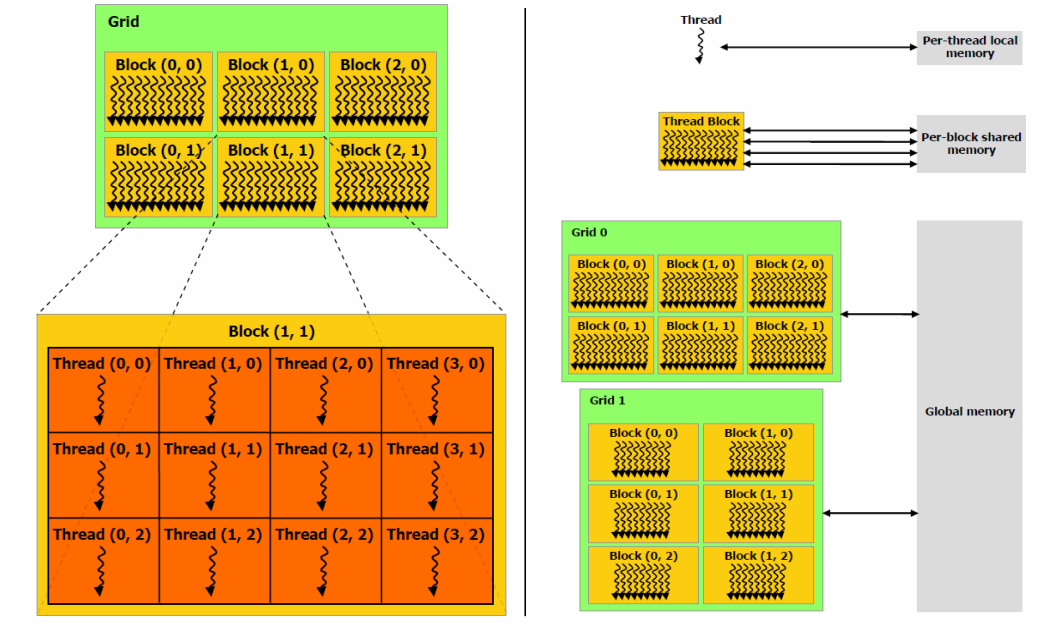

Figura 2.20. Jerarquía de hilos (izquierda) y de memoria (derecha en CUDA. Fuente: [11].

2.7.2 OPENCL (OPEN COMPUTING LANGUAGE)

OpenCL (*Open Computing Language*) es un estándar para programación de propósito general multiplataforma. Al igual que CUDA, OpenCL permite codificar algoritmos para GPU utilizando un lenguaje basado en C99. La mayor diferencia con CUDA es que OpenCL es abierto y por lo tanto puede trabajar con cualquier hardware compatible, incluidas las GPU nVidia.

OpenCL no está pensado para trabajar solamente con GPU's, sino con cualquier hardware compatible, como procesadores multinúcleo. OpenCL incluye una amplia variedad de funciones integradas y se ajusta totalmente al estándar IEEE 754. Al contrario que CUDA en OpenCL los *kernels* pueden ser pre-compilados o compilados en tiempo de ejecución (compilación JIT) a elección del desarrollador.

Debido a la multitud de dispositivos que son compatibles con OpenCL y la heterogeneidad de estos, los *kernels* pueden idearse tanto para proporcionar paralelismo de datos, lo que coincide con la arquitectura de GPUs, o tareas en paralelo, pensadas para la arquitectura de las CPUs.

OpenCL define un modelo de memoria multinivel con rango de memoria que oscila desde la memoria privada, visible sólo para las unidades de cálculo individual en el dispositivo, hasta la memoria global, visible para todas las unidades de cálculo del dispositivo. Según el subsistema de la memoria actual, se permite que diferentes espacios de memoria funcionen a la vez.

OpenCL define 4 espacios de memoria: privada, local, constante y global. La Figura 2.21 muestra un diagrama de la jerarquía de la memoria definida por OpenCL.

La memoria privada es la que puede usarse únicamente por una unidad de cálculo única. Esto es similar a los registros en una única unidad de cálculo o un núcleo CPU único.

La memoria local es la memoria que puede usarse por los elementos de trabajo en un grupo de trabajo.

La memoria constante es la memoria que puede usarse para almacenar datos constantes para acceso sólo de lectura por todas las unidades de cálculo de un dispositivo durante la ejecución de un *kernel*. El procesador anfitrión es responsable de asignar e iniciar los objetos de memoria que residen en este espacio de memoria.

Finalmente, la memoria global es la que puede usarse por todas las unidades de cálculo de un dispositivo. [12]

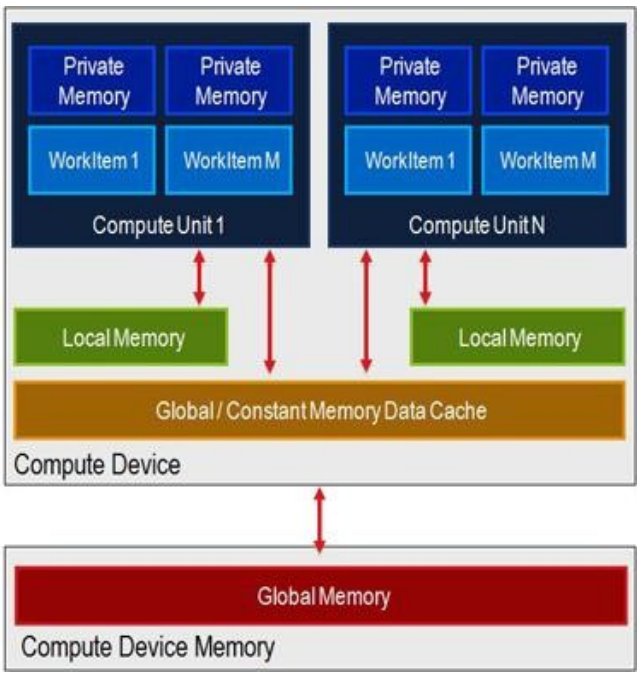

Figura 2.21. Memoria en OpenCL.

2.8 C++

C++ es un lenguaje de programación diseñado a mediados de los 80 por Bjarne Stroustrup. Es un lenguaje multiparadigma que permite la manipulación de objetos. Desde su creación, ha sido definido por diferentes estándares ISO/IEC. La última revisión del estándar se aprobó en 2011 dando lugar a la versión ISO/IEC 14882:2011. Se puede encontrar más información sobre este estándar en [13].

C++ incluye varias características que cabe destacar debido a su importancia y utilidad para el programador. Algunas de ellas han sido añadidas en el último estándar y otras ya existían en versiones anteriores del lenguaje.

2.8.1 PROGRAMACIÓN GENÉRICA CON *TEMPLATES*

Los *templates* o plantillas son el mecanismo que utiliza C++ para implementar programación genérica. Permiten que una clase en su definición trabaje con datos abstractos especificándole más adelante que tipos de datos concretos se utilizarán. Esto permite reutilizar una gran cantidad de código y centrar la programación en los algoritmos en vez de en los datos.

```
/**
* La linea siguiente define que cada vez que aparezca "t"
* se modificará el dato por otro que más adelante definiremos.
*/
template <class t> class prueba{
    private:
    /**
    * El array se define como un tipo genérico t, que más adelante
    * se sutituirá por el tipo de dato que se utilice
    */
        t array[5];
        int poscicion;

    public:
        prueba(){}
        virtual ~prueba();
        bool ingresar_datos(t & datos);
      //La función devuelve un tipo genérico t
        t mostrar_datos(int & pos);
};
```

Figura 2.22. Implementación de *template* en C++.

```
int main (){
      int   dato1 = 1;
      char  dato2 = 'a';
      float dato3 = 3.14;
      int   pos = 0;

      prueba<float>prueba3;
      prueba<char>prueba2;
      prueba<int>prueba1;

      prueba1.ingresar_datos(dato1);
      cout << prueba1.mostrar_datos(pos) << endl;

      prueba2.ingresar_datos(dato2);
      cout << prueba2.mostrar_datos(pos) << endl;

      prueba3.ingresar_datos(dato3);
      cout << prueba3.mostrar_datos(pos) << endl;
      return 0;
```

Figura 2.23. Instanciación y uso de objeto definido con *template* en C++

En la Figura 2.22 y la Figura 2.23 se muestra la definición y el uso de un *template*. La primera de las figuras muestra su definición. Se aprecia como el tipo del array es de tipo `t`, un tipo genérico que sustituirá por un tipo completo en la instanciación de cada objeto de la clase, lo mismo ocurre en el parámetro `datos` de la función `ingresar_datos` y con el valor de retorno de la función `mostrar_datos`.

La segunda figura muestra la instanciación y uso de los objetos definidos como una plantilla. En la declaración de los objetos tipo `prueba` se indica, entre símbolos "<" y ">" el tipo de dato para el que se instanciará la plantilla. Este tipo será el que sustituirá al tipo `t` genérico indicado en la definición de la plantilla. Posteriormente cada objeto tipo `prueba` se puede utilizar como cualquier otro tipo definido por el usuario, teniendo en cuenta que el tipo de los parámetros de las funciones miembro y los retornos de estas dependen del tipo para el que se haya instanciado la plantilla.

2.8.2 COMPROBACIONES EN TIEMPO DE COMPILACIÓN (*STATIC_ASSERTS*)

Los *static_asserts* o comprobaciones en tiempo de compilación son una característica introducida en el estándar C++11 que permite hacer comprobaciones en tiempo de compilación provocando que la compilación termine si la evaluación de la comprobación resulta ser falsa.

La utilización de un *static_assert* sigue la siguiente estructura:

```
static_assert(expression,message);
```

El compilador evalúa la expresión booleana definida por el parámetro `expression` y en caso de resultar falsa muestra el mensaje definido por el parámetro `message` y termina la compilación.

El uso de *static_assert* permite detectar errores en tiempo de compilación haciendo así más sencilla la posterior depuración del código.

2.8.3 CARACTERÍSTICAS DE TIPOS (*TYPE_TRAITS*)

Una de las novedades de C++11 es la inclusión de los denominados *type_traits*. Los *type_traits* permiten consultar o modificar en tiempo de compilación las propiedades de los tipos de datos.

Existen numerosos *type_traits* para comprobar diferentes características, como por ejemplo si un tipo es numérico o no, si se puede copiar, si es constante, si tiene signo, etc. Otro tipo de *type_traits* permite modificar algunas características de los tipos en tiempo de compilación como por ejemplo convertir un valor con signo en un valor sin él. Se puede encontrar más información sobre los diferentes *type_traits* implementados en C++11 en [14].

2.8.4 SEMÁNTICA DE MOVIMIENTO

La semántica de movimiento es una mejora introducida con el nuevo estándar C++11. La semántica de movimiento permite incrementar el rendimiento de las aplicaciones disminuyendo las copias profundas de datos en memoria.

Cuando un objeto es copiado, ya sea a través del operador de asignación o su constructor de copia, utilizando semántica de movimiento, lo único que se copia es el puntero a los datos en memoria, nunca los propios datos. De esta forma el tiempo que se tarda en

copiar un objeto es mucho menor que en el caso de hacer una copia profunda de los datos. Esta disminución de tiempo es más apreciable cuanto mayor sea la cantidad de datos que componen el objeto en memoria, ya que la copia del puntero toma un tiempo constante independientemente de la cantidad de datos o la zona de memoria que esté apuntando. Una explicación más detallada de la semántica de movimiento se encuentra en la sección 5.1.1 de este documento.

2.8.5 FUNCIONES LAMBDA

Otra de las capacidades que implementa C++11 es la de construir funciones anónimas, también conocidas como funciones lambda o *closures.* Una función lambda es una función que se puede escribir en línea (por lo general para pasar como parámetro a otra función con nombre, similar a la idea de *functor* o de puntero a función).

```
class AddressBook{
    public:
    // Usar una plantilla permite ignorar la diferencia entre
    // functors, lambdas y punteros a función
    template<typename Func>
    std::vector<std::string> findMatchingAddresses (Func func)
    {
        std::vector<std::string> results;
        for ( auto itr = _addresses.begin(), end = _addresses.end();
        itr != end; ++itr )
        {
            // Llamada a la función que se recibe como parámetro
            if ( func( *itr ) )
            {
                results.push_back( *itr );
            }
        }
        return results;
    }
    private:
    std::vector<std::string> _addresses;
};
```

Figura 2.24. Clase que utiliza una función lambda para realizar una búsqueda en un vector.

```
AddressBook global_address_book;
vector<string> findAddressesFromOrgs ()
{
    return global_address_book.findMatchingAddresses(
        // Declaración de function lambda
        [] (const string& addr) { return addr.find( ".org" ) !=
        string::npos; }
    );
}
```

Figura 2.25. Utilización de función lambda para definir el comportamiento de la búsqueda.

En la Figura 2.24 y la Figura 2.25 se muestra un ejemplo de uso de función lambda. La primera de las figuras muestra una clase que implementa un método que busca en un vector. La función miembro `findMatchingAddresses` recibe como parámetro una función lambda. El resultado de la aplicación de esta función sobre cada elemento del vector decide si el elemento coincide con la búsqueda o no.

En la segunda figura se invoca la función miembro `findMatchingAddresses` de la clase anterior pasándole como parámetro una función lambda. Esta función lambda comprueba si la cadena que recibe como parámetro contiene la subcadena ".org", en caso positivo devuelve "true" y si no "false".

El resultado final es que el método `findMatchingAddresses` evaluará el resultado de la función lambda con cada elemento del vector `_addresses` y si el elemento contiene la cadena ".org" el retorno de la función lambda provocará que se ejecute el código dentro de la sentencia `if`.

Como se puede observar si más adelante se quisiese modificar el patrón de búsqueda no sería necesario modificar y volver a compilar la clase `AddressBook`, sino que con invocar el método `findMatchingAddresses` con otra función lambda cuyo cuerpo implementase la condición de búsqueda deseada sería suficiente.

2.8.6 STL (STANDARD *TEMPLATE* LIBRARY)

La biblioteca estándar de plantillas es una biblioteca que provee al programador de estructuras de datos, algoritmos, iteradores, etc. No es la única biblioteca de este tipo para C++, pero si la única estándar. La mayor parte de los compiladores de C++ disponen (o disponían) de bibliotecas similares y, también, están disponibles varias bibliotecas comerciales. Uno de los problemas de estas bibliotecas es que son mutuamente incompatibles, lo que obliga a los programadores a aprender nuevas bibliotecas y a migrar de un proyecto a otro y de un compilador a otro. Sin embargo, STL ha sido adoptado por el comité ANSI de estandarización del C++, lo que significa que está soportado como una extensión más del lenguaje por todos los compiladores.

2.8.7 SOPORTE A THREADS Y A TIPOS ATÓMICOS

El estándar C++11 define una biblioteca que da soporte a la programación con hilos para plataformas multiprocesador o multihilo. Esta biblioteca ofrece al programador todas las herramientas necesarias para poder generar código multihilo utilizando hilos nativos.

También implementa tipos atómicos y sus correspondientes operaciones atómicas como parte de la biblioteca estándar. Se puede encontrar más información sobre threads y tipos atómicos en C++11 en [14].

2.9 HERRAMIENTAS DE *PROFILING*

Una de las etapas del desarrollo de un programa consiste en optimizar su rendimiento. Para llevar a cabo esta tarea es necesario saber en qué partes del programa se concentra la mayor parte del tiempo de ejecución. Para realizar esta labor se utilizan las herramientas de *profiling* que controlan la ejecución del programa sobre el que se quiere obtener información y toman datos de la misma que después pueden ser analizados de forma gráfica.

Existen varias formas de realizar mediciones de rendimiento del código. Una de ellas es la simulación. Esta técnica se basa en introducir información adicional en el código, lo que se conoce como instrumentación, para después hacer una ejecución simulada del mismo y obtener datos de rendimiento, como por ejemplo que funciones son más utilizadas, cuales tardan más en ejecutar o cómo accede el programa a memoria. Este método es muy simple y se pueden encontrar varias utilidades que lo utilizan. Las simulación tiene el inconveniente de que es muy lenta debido a que la instrumentación genera gran sobrecarga. Además dado que habitualmente las zonas sobre las que se quiere medir el rendimiento suelen estar anidadas ocurre que el código adicional para medir rendimiento en las zonas internas hace que la ejecución de las zonas externas sea más lenta, por lo que hay que analizar los resultados cuidadosamente.

Una forma de realizar medidas de rendimiento sin que el código adicional introduzca una sobrecarga es la utilización de contadores hardware. Estos elementos están incluidos en algunos procesadores modernos y son contadores que se incrementan cada vez que se produce un evento como un acceso a memoria o un fallo de caché. Existen varias técnicas como el muestreo temporal (Time Based Sampling o TBS) y el muestreo de eventos (Time Based Sampling o EBS) que hacen uso de los contadores hardware para recabar información de comportamiento del código[15].

2.9.1 CACHEGRIND Y KCACHEGRIND

Callgrind es un simulador que utiliza la estructura de instrumentación de Valgrind por lo que no es necesario recurrir a contadores hardware que no están presentes en todas las máquinas y a que analiza los binarios sin necesidad de modificarlos. Con Callgrind se puede simular sólo las partes que interesan del código, minimizando de esta manera la ralentización de la simulación.

Cachegrind es capaz de simular la caché capturando los accesos a la misma. La información recogida incluye el número de instrucción y los fallos de caché de primer y segundo nivel y los relaciona con la región de código correspondiente. A continuación, combinando estos datos con las latencias de fallo en los procesadores más comunes, es posible obtener una estimación del tiempo empleado.

Otra de las funciones de Cachegrind a través del complemento Callgrind es la creación del árbol de llamadas de las funciones, es decir, cómo las funciones se llaman entre sí y qué eventos ocurren mientras se ejecutan. También es posible separar la información por hilos o por contextos de llamadas.

Cachegrind genera ficheros de salida con toda la información recabada, sin embargo es muy difícil analizar la información directamente de estos ficheros. Para solventar este inconveniente se utiliza un entorno gráfico llamado KCachegrind. KCachegrind es una herramienta de visualización de información de rendimiento que permite ver la información generada por Cachegrind de forma ordenada, mostrando las funciones ordenadas por sus costes inclusivos y exclusivos o agrupadas por fichero fuente, biblioteca o clase. Además permite varios métodos de visualización para una función determinada como:

- Un gráfico de llamadas que muestra la sección que rodea a la función seleccionada.
- Un mapa en forma de árbol que permite ver las relaciones entre funciones anidadas con datos sobre el coste computacional que producen, permitiendo detectar funciones muy pesadas rápidamente.
- Vista del código fuente y anotaciones del desensamblador con información del coste computacional de cada línea e instrucciones de ensamblador.

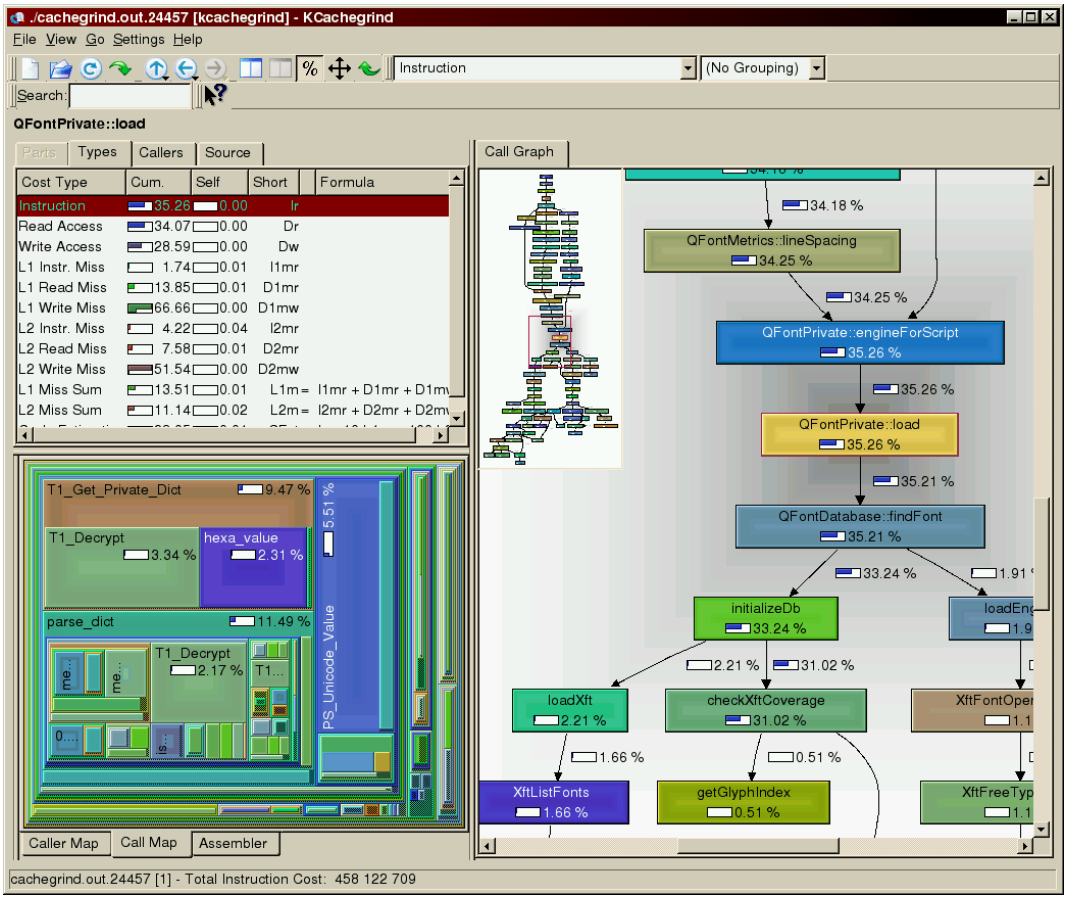

Figura 2.26. Vista general de KCachegrind.

En la Figura 2.26 se observa la interfaz de KCachegrind, la parte derecha de la figura muestra el árbol de llamadas entre funciones, cada función se representa con un nodo y en él se indica el porcentaje de tiempo sobre el total que ha utilizado. Para cada función este porcentaje es la suma del porcentaje de tiempo que tardan en ejecutar todas las funciones a las que se llama y del porcentaje de tiempo que tarda la propia función. Entre cada par de nodos relacionados, unidos a través de una flecha, se representa el porcentaje de tiempo acumulado por todas las funciones que quedan por debajo de ese punto. Si se selecciona un nodo en el grafo los porcentajes de tiempo se reajustan en los nodos que dependen de él, es decir, supongamos una función C que es llamada por las funciones A y B, si seleccionamos el nodo que representa la función A el porcentaje de tiempo del nodo de la función C sólo dará el porcentaje de tiempo empleado en la función C cuando es llamada por la función A y no incluirá el tiempo de las llamadas de la función B.

En la parte superior izquierda de la figura se muestra el resumen de los accesos a memoria de la función seleccionada en el grafo de la izquierda. Esta parte incluye datos sobre

los accesos a memoria de lectura de datos, escritura de datos, lectura de instrucciones, fallos y acierto de caché de datos e instrucciones de nivel 1 y 2 y totales de cada nivel.

La parte inferior derecha de la imagen muestra un mapa de llamadas. Este mapa de llamadas representa lo mismo que el grafo de la parte derecha aunque de una forma menos clara por lo que no se utiliza apenas.

2.9.2 INTEL® VTUNE® PERFORMANCE ANALYZER

VTune® es una herramienta de *profiling* para arquitecturas Intel que permite la visualización y análisis de datos generado en tiempo de ejecución. VTune permite obtener datos tanto de código que está ejecutando en el propio sistema como de código que está ejecutando sobre una máquina virtual. La comunicación de los datos de rendimiento generados y el analizador se realiza a través de una biblioteca estática que recibe los datos generados por la aplicación que se ejecuta y los envía al analizador VTune, cargando previamente un objeto de una biblioteca dinámica.

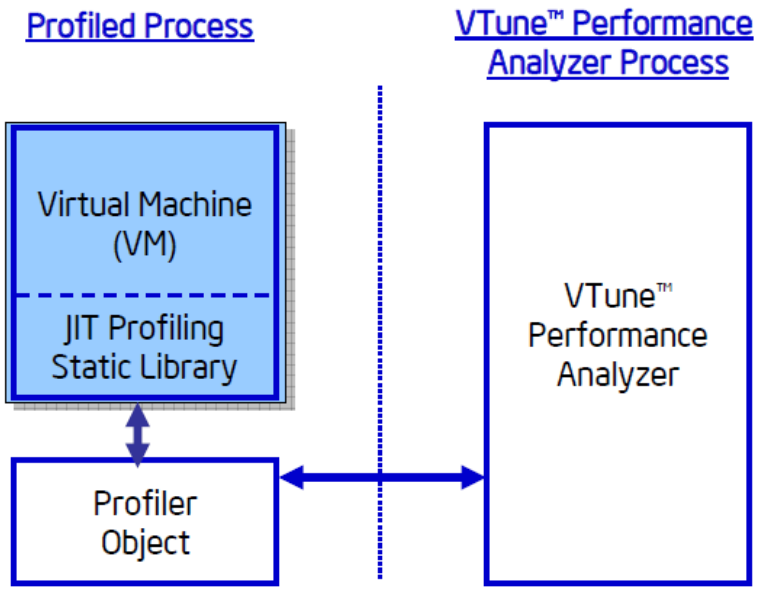

Figura 2.27. Comunicación entre el código en ejecución y VTune. Fuente[16].

VTune utiliza la técnica EBS *Events Based Sampling* la cual produce una sobrecarga muy pequeña. Cada cierto tiempo se produce una interrupción que VTune utiliza para recoger los valores de ciertos registros del procesador que describen el entorno de ejecución, como por ejemplo los ticks de reloj transcurridos para medir el tiempo que tarda en ejecutarse una función o el numero de fallos y aciertos de caché para comprobar si se está accediendo de forma eficiente a la memoria. Recogiendo además el valor del contador de programa en el momento de la interrupción VTune puede asociar la información de los contadores con el

código que se está ejecutando y, si el código fuente está disponible VTune es capaz de asociar esta información con él.

Esta técnica produce una sobrecarga muy pequeña sobre el código ya que la instrumentación del mismo es muy pequeña. Realizando 1000 interrupciones por segundo para recoger datos se produce una sobrecarga de alrededor del 1% en el sistema con una precisión bastante alta[17]. Hay que tener en cuenta que la precisión de las medidas depende del procesador que se utilice y del número de contadores hardware que éste tenga.

Al igual que KCacheGrind, VTune también ofrece información sobre el árbol de llamadas a funciones, mostrando la jerarquía de funciones, tiempo consumido por cada una de ellas y sus descendientes y número de llamadas.

Otra de las funciones de VTune es el análisis del código fuente para detectar los problemas de rendimiento causados por el compilador como valores retornados ignorados, conversiones de tipos, etc. que pueden afectar al rendimiento del programa y que no pueden optimizarse por el compilador debido a que las especificaciones del lenguaje no lo permiten. Usando la propia tecnología de los compiladores, VTune es capaz de detectar estos problemas y dar indicaciones sobre optimizaciones posibles en una línea de un tramo de código, como por ejemplo desenrollar un bucle.

	Muestreo	**Instrumentación**
Sobrecarga	Muy baja, alrededor del 1%.	Muy alta, entre 10% y 500%.
Análisis de todo el sistema	Si, análisis de las aplicaciones, drivers y funciones del S.O.	No, sólo la aplicación y su árbol de llamadas.
Detección de eventos inesperados.	Si, capacidad de detectar que otro programas se ejecutan y como afectan al rendimiento.	No, sólo la propia aplicación.
Inicialización	No necesaria.	Inserción de código adicional en el ejecutable (instrumentación).
Datos recogidos	Datos de contadores, estado del procesador y del S.O.	Gráfico de llamadas, jerarquía de funciones, número de llamadas.
Granularidad de los datos	Instrucciones de ensamblador y línea que causó un evento.	Funciones.
Detección de problemas en algoritmos	No, limitado a procesos, hilos, módulos y funciones.	Si, puede comprobar que un algoritmo un camino a través de la aplicación es más caro computacionalmente.

Tabla 2.2 Comparativa entre técnica de muestreo y de simulación. Fuente: [17]

3 METODOLOGÍA

En este capítulo se explica de forma resumida el proceso que se ha seguido en la elaboración del proyecto.

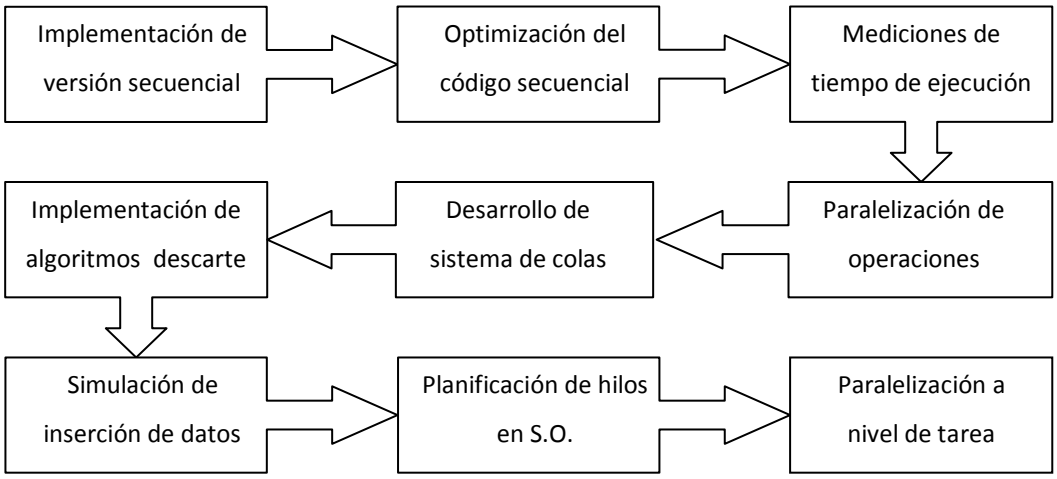

Figura 3.1 Fases del desarrollo

3.1 CÓDIGO SECUENCIAL DE PARTIDA

Para llevar a cabo el proyecto se ha implementado una versión secuencial del algoritmo de los filtros de Kalman en C++. Esta versión secuencial incluye toda la funcionalidad necesaria para llevar a cabo tanto las operaciones algebraicas con matrices como las diferentes fases que componen el algoritmo de los filtros de Kalman, además, también ofrece la funcionalidad necesaria para obtener los datos necesarios desde el disco así como para almacenar los resultados en él o presentarlos por pantalla.

La primera versión secuencial del código está compuesta de 2 clases, implementadas siguiendo el estándar ISO/IEC 14882:2011, también conocido como C++11, publicado en septiembre de 2011. La elección de este estándar viene motivada principalmente por la gran cantidad de optimizaciones que implementa.

La primera de las clases, *KFMatrix*, representa el concepto de matriz e implementa la funcionalidad necesaria para llevar a cabo operaciones algebraicas con matrices, así como las funcionalidades que implican movimiento de datos, tales como carga y almacenamiento de datos a disco.

La segunda clase, *KFLib*, representa las diferentes fases del algoritmo de los filtros de Kalman que se ha descrito en el punto 2.1.1 e implementa la lógica necesaria para aplicarlo a los datos. Esta clase está compuesta de objetos *KFMatrix*, ya que como se ha podido ver en el capítulo 2 de este documento, el algoritmo de los filtros de Kalman se basa en la aplicación de operaciones algebraicas sobre una serie de datos dados en forma matricial.

3.2 OPTIMIZACIÓN DEL CÓDIGO SECUENCIAL

El último paso de la versión secuencial del algoritmo ha sido optimizarlo para conseguir una ejecución lo más barata posible tanto en términos de tiempo como de recursos del sistema. Las optimizaciones que se han aplicado al código secuencial se detallan en el punto 5.1 de este documento.

3.3 MEDICIÓN DE TIEMPOS DE EJECUCIÓN

El primer paso tras obtener una versión secuencial optimizada ha sido intentar disminuir el tiempo de cómputo de las operaciones más costosas del algoritmo, ya que optimizar las partes más lentas del algoritmo producirá una mayor aceleración sobre el total que optimizando partes menos lentas. Para poder identificar que operaciones eran las que más tiempo de ejecución suponían se han utilizado herramientas de *profiling*. Como se ha explicado en el punto 2.9. Estas herramientas permiten conocer en detalle cómo se ejecuta un proceso, en que funciones consume más tiempo, más memoria, etc.

3.4 PARALELIZACIÓN DE OPERACIONES ALGEBRAICAS

Una vez identificadas las funciones que más tiempo consumen se ha llevado a cabo la paralelización de las mismas utilizando dos técnicas diferentes, OpenMP y ArBB, con el fin de acelerar su ejecución.

3.5 IMPLEMENTACIÓN DE UN SISTEMA DE COLAS PARA EL SEGUIMIENTO DE VARIOS ELEMENTOS SIMULTÁNEOS

El siguiente paso ha sido la implementación de un mecanismo que permitiese aplicar el algoritmo de los filtros de Kalman a varios objetos móviles simultáneos. Para conseguir esto se ha diseñado e implementado un sistema gestor de tareas basado en colas ordenadas. Este gestor permite tratar los datos de mediciones de todos los elementos de forma que ninguno de ellos monopolice el uso del sistema.

Este sistema gestor de tareas está implementado en 3 clases C++11. La primera de ellas, *KFData*, representa un dato de medición de un objeto. Está compuesta por un objeto *KFMatrix* que almacena la medición en forma de matriz y otros elementos necesarios para la gestión del dato como una marca de tiempo y un identificador numérico del objeto al que pertenece la medición.

Las medidas tomadas de cada objeto se almacenan de forma ordenada en colas. Estas colas se implementan en la clase *KFQueue*. La clase *KFQueue* está compuesta de un contenedor que almacena objetos de tipo *KFData* con las distintas medidas que se obtienen de un objeto y un identificador que coincide con el del objeto y por tanto con el de las medidas. La clase implementa la lógica necesaria para hacer uso de la cola, como funciones para insertar elementos, extraerlos, comprobar el tamaño de las colas, etc.

Finalmente la clase *KFTaskManager* se encarga de la gestión de las colas de tareas así como de los resultados que se van generando durante la ejecución. Esta clase está formada por una cola circular sobre la que se va iterando para seleccionar el siguiente objeto sobre el que se aplicará el algoritmo, de esta forma el tiempo de procesamiento se reparte de forma equitativa entre todos los objetos controlados. La clase también almacena en un contenedor de objetos de tipo *KFLib* los resultados que se generan.

3.6 IMPLEMENTACIÓN DE ALGORITMOS DE DESCARTE

Debido que no es posible en todos los caso tratar todos los datos que se obtienen se hace necesario encontrar una forma de decidir qué datos deben tratarse y cuales deben descartarse en caso de saturación del sistema.

Se han implementado varios algoritmos de descarte de datos para poder decidir cuál funciona mejor y en qué situaciones.

3.7 SIMULACIÓN DE INSERCIÓN DE DATOS EN TIEMPO REAL

Se ha implementado un sistema que permite simular la llegada de datos desde los sensores en tiempo real. Este sistema permite cargar los datos desde un fichero a memoria e irlos insertando en el momento adecuado en las colas donde se recibirían los datos de los sensores.

3.8 AJUSTE DE PARÁMETROS DE PLANIFICACIÓN DE HILOS DE S.O

Debido a que las diferentes tareas tienen distinta prioridad y algunas deben cumplir, en cierta medida, plazos temporales se ha llevado a cabo un análisis de las necesidades de cada tarea que debe realizar el sistema y se han ajustado los parámetros de planificación de las mismas en S.O de forma que se ajusten lo máximo posible a sus requerimientos.

3.9 PARALELIZACIÓN A NIVEL DE TAREA

El siguiente paso que se ha llevado a cabo ha sido paralelizar la gestión de tareas. Con este paso se consigue que varios hilos puedan ejecutar el algoritmo de los filtros de Kalman sobre varios datos de objetos móviles al mismo tiempo.

La paralelización de tareas se ha llevado a cabo de varias formas. Por una parte se ha implementado una versión basada en threads nativos de C++. Por otro lado se han desarrollado varias versiones utilizando la biblioteca de paralelización Intel Threading Building Blocks. Con esta biblioteca se han implementado 3 versiones diferentes del sistema utilizando diferentes mecanismos de paralelización de tareas.

3.10 EVALUACIÓN

En esta fase se han realizado una serie de pruebas que han tenido como objeto una evaluación del rendimiento, ventajas, inconvenientes, etc. de las diferentes versiones paralelas implementadas.

3.11 ANÁLISIS DE LOS RESULTADOS

Finalmente se ha realizado un análisis de los resultados obtenidos con cada implementación del sistema con el fin, tanto de encontrar en qué partes del mismo cabe una posibilidad de mejora, como de abrir una puerta a posibles trabajos futuros en el mismo ámbito o similares.

4 ANÁLISIS Y DISEÑO

4.1 ALCANCE DEL SOFTWARE

El sistema software que se desarrollará será un sistema de tiempo real no crítico de calidad de servicio, capaz de aplicar el algoritmo del filtro de Kalman lineal a los datos de medidas de objetos móviles.

4.2 CASOS DE USO

El sistema que se va a diseñar funcionaria como sistema intermedio entre los sensores que tomasen las medidas de los objetos móviles y otro sistema externo que obtuviese los resultados generados, por lo tanto sólo existen dos casos de uso que se muestran a continuación.

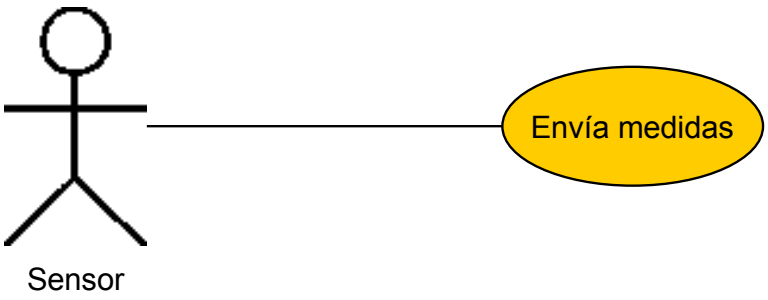

Figura 4.1. Caso de uso I.

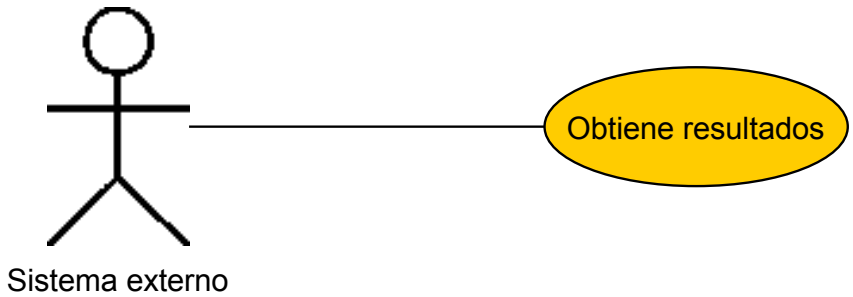

Figura 4.2. Caso de uso II.

4.3 REQUISITOS

El presente punto especifica los requisitos de usuario y software del sistema que se desarrolla. Los objetivos de este proceso de especificación son identificar, validar y documentar los requisitos de usuario y software, es decir, determinar las características que deberán tener el sistema o las restricciones que deberá cumplir para que sea aceptado por los futuros usuarios de la aplicación.

4.3.1 REQUISITOS DE USUARIO

En este punto se enumeran los requisitos de usuario que se ha identificado.

4.3.1.1 REQUISITOS DE CAPACIDAD

ID	Descripción	Prioridad
RU-01	El sistema deberá poder aplicar el algoritmo a varios objetos móviles al mismo tiempo.	Muy alta
Necesidad	**Fuente**	**Verificable**
Muy alta	Usuario	SI

ID	Descripción	Prioridad
RU-02	El sistema deberá poder trabajar con cualquier tipo de dato siempre que este sea de coma flotante	Muy alta
Necesidad	**Fuente**	**Verificable**
Muy alta	Usuario	SI

ID	Descripción	Prioridad
RU-03	El sistema deberá poder descartar los datos que no pueda tratar sin llegar a colapsarse	Muy alta
Necesidad	**Fuente**	**Verificable**
Muy alta	Usuario	SI

ID	Descripción	Prioridad
RU-04	El sistema debe adaptar su rendimiento a las prestaciones de la máquina	Muy alta
Necesidad	**Fuente**	**Verificable**
Muy alta	Usuario	SI

4.3.1.2 REQUISITOS DE RESTRICCIÓN

ID	Descripción	Prioridad
RU-04	El código debe ser estándar y portable	Media
Necesidad	**Fuente**	**Verificable**
Media	Usuario	SI

ID	Descripción	Prioridad
RU-05	El sistema debe estar desarrollado en su totalidad en lenguaje C++	Media
Necesidad	**Fuente**	**Verificable**
Media	Usuario	SI

ID	Descripción	Prioridad
RU-06	El software debe ser de código abierto	Media
Necesidad	**Fuente**	**Verificable**
Media	Usuario	SI

4.3.2 REQUISITOS SOFTWARE

En este punto se detallan los requisitos software que se ha derivado de los requisitos de usuario.

4.3.2.1 REQUISITOS FUNCIONALES

ID	Descripción	Prioridad
RS-01	El sistema debe poder diferenciar los diferentes objetos móviles que se están gestionando	Muy alta
Necesidad	**Fuente**	**Verificable**
Muy alta	Usuario	SI

ID	Descripción	Prioridad
RS-02	El sistema deberá poder ordenar los datos que recibe de cada objeto móvil	Muy alta
Necesidad	**Fuente**	**Verificable**
Muy alta	Usuario	SI

ID	Descripción	Prioridad
RS-03	El sistema deberá aprovechar en la capacidad de cómputo de los sistemas multinúcleo y/o multiprocesador.	Alta
Necesidad	**Fuente**	**Verificable**
Alta	Usuario	SI

ID	Descripción	Prioridad
RS-04	La implementación se llevará a cabo con plantillas de C++	Alta
Necesidad	**Fuente**	**Verificable**
Alta	Usuario	SI

ID	Descripción	Prioridad
RS-05	Se diseñará un algoritmo que descarte los datos que el sistema no sea capaz de tratar	Alta
Necesidad	**Fuente**	**Verificable**
Alta	Usuario	SI

ID	Descripción	Prioridad
RS-06	La implementación se realizará utilizando funciones bibliotecas estándar de C++	Alta
Necesidad	**Fuente**	**Verificable**
Alta	Usuario	SI

ID	Descripción	Prioridad
RS-07	Todas la bibliotecas de terceros que se utilicen deberán ser portables	Alta
Necesidad	**Fuente**	**Verificable**
Alta	Usuario	SI

ID	Descripción	Prioridad
RS-08	Todas la bibliotecas de terceros que se utilicen deberán ser de código abierto	Alta
Necesidad	**Fuente**	**Verificable**
Alta	Usuario	SI

4.4 CLASES DESARROLLADAS

El sistema consta de dos subsistemas diferenciados. El primero se encarga de la realización de las operaciones necesarias para la aplicación del algoritmo del filtro de Kalman discreto. El segundo subsistema ofrece un sistema de gestión de tareas para poder aplicar el algoritmo sobre los datos de más un objeto móvil al mismo tiempo. Además de estos dos subsistemas también se ha implementado una clase auxiliar para simular la llegada de datos en tiempo real.

4.4.1 ALGORITMO DEL FILTRO DE KALMAN

Esta parte del sistema está compuesto por dos clases, `KFMatrix` y `KFLib`. La primera de estas dos clases representa una matriz e implementa las operaciones algebraicas necesarias para trabajar con ella en el ámbito del algoritmo del filtro de Kalman. La segunda representa el propio algoritmo de los filtros de Kalman e implementa las funciones que permiten calcular los diferentes pasos del mismo. Ambas clases son plantillas que reciben un solo parámetro, el cual indica el tipo de dato con el que se trabajará, éste debe ser un tipo de coma flotante.

4.4.1.1 CLASE KFMATRIX

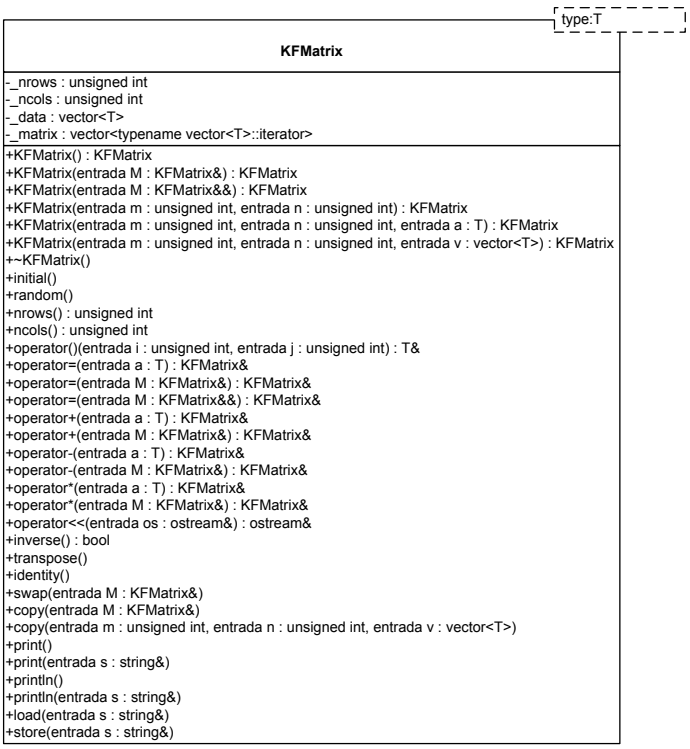

Figura 4.3. Clase *KFMatrix*.

Como se puede observar en la Figura 4.3 la clase `KFMatrix` está compuesta por dos enteros sin signo que almacenan el número de filas y columnas de la matriz. Los elementos de la matriz se almacenan en el atributo `_data`, el cual es de tipo `vector<T>`, siendo `T` el parámetro de plantilla de la clase. El atributo `_matrix` almacena punteros al primer valor de cada fila, pudiendo de esta manera acceder a los elementos de la matriz utilizando la notación `[i][j]` que es más legible.

4.4.1.2 CLASE KFLIB

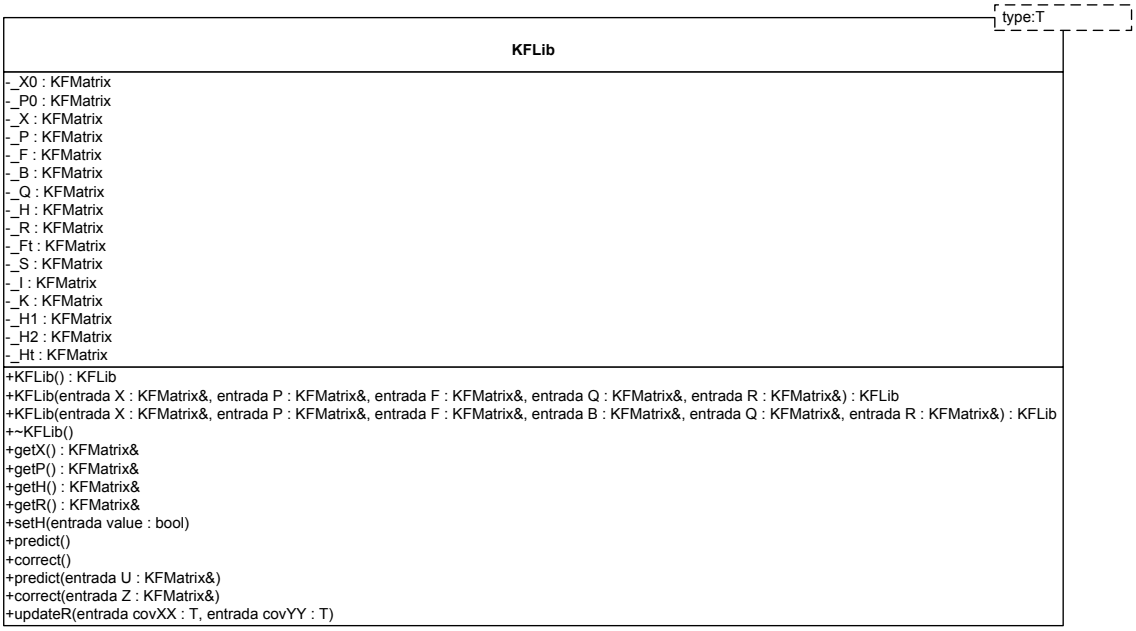

Figura 4.4. Clase *KFLib.*

La clase `KFLib`, implementa las funciones necesarias para aplicar el algoritmo de los filtros de Kalman. Sus atributos son objetos del tipo `KFMatrix` que representan cada una de las matrices que se utilizan en el algoritmo.

4.4.2 GESTIÓN DE TAREAS

El segundo subsistema es el encargado de gestionar cuándo se tratan los datos en caso de coexistencia de varios objetos móviles. Además, permite hacerlo de forma concurrente con varios hilos. Este subsistema está formado por 3 clases implementadas en C++, `KFData`, `KFQueue`, `KFTaskManager`.

Para poder gestionar los datos que se obtienen de cada objeto controlado se ha diseñado una serie de colas capaces de almacenar los datos que se reciben. Cada una de estas colas recibe y almacena los datos de un mismo objeto.

Las colas de datos son tratadas de forma alterna, tomando el primer dato de una cola, tratándolo y pasando a la siguiente cola, así hasta volver al principio y repetir el proceso. En el caso en que una cola este vacía se salta a la siguiente. Para llevar a cabo este proceso indefinidamente se ha implementado una cola circular de punteros a las colas de datos. Esta cola funciona de tal forma que, cuando se quiere obtener un dato de una de las colas de datos se extrae el puntero a la misma y no se vuelve a introducir hasta que se ha terminado de realizar el tratamiento del dato. De esta forma se minimizan los bloqueos, ya que un hilo sólo bloqueará la cola durante el tiempo suficiente para extraer o introducir un puntero en ella. Además este sistema permite asegurar que en ningún caso se están tratando dos datos de la misma cola al mismo tiempo.

La clase `KFData` representa una medición de posición sobre un objeto móvil. Cada una de estas mediciones se agrupa en colas ordenadas representadas por instancias de la clase `KFQueue`. Finalmente la clase `KFTaskManager` gestiona el acceso a cada cola y almacena los resultados que se van generando.

4.4.2.1 CLASE KFDATA

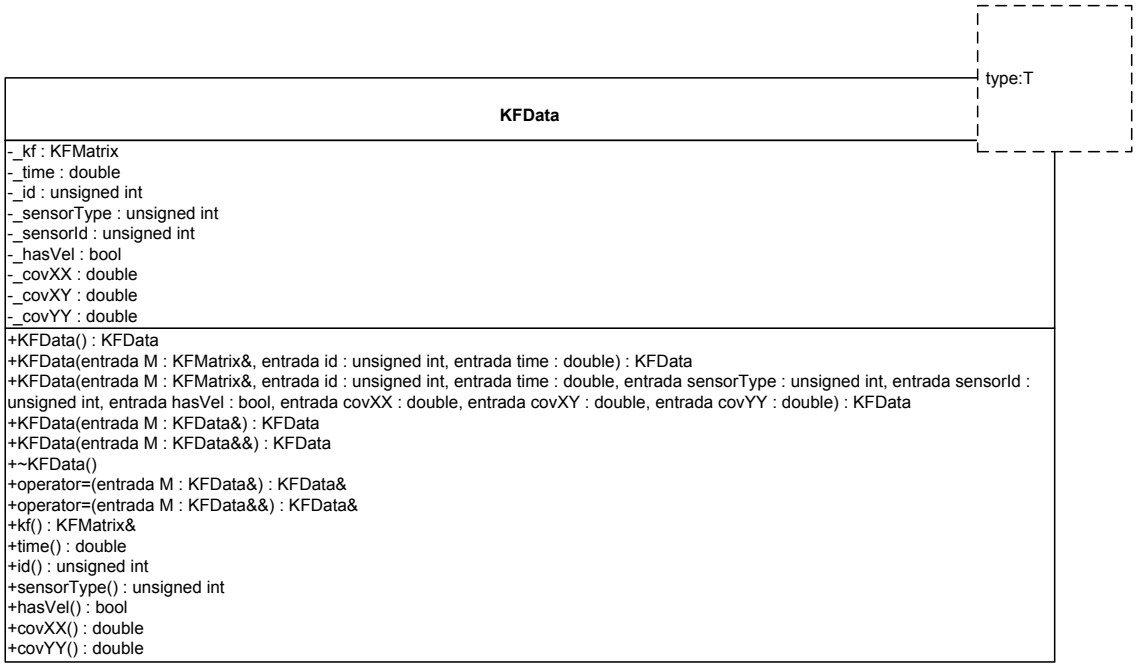

Figura 4.5. Clase *KFData.*

La clase `KFData` representa la medición de posición en un instante determinado de un objeto móvil. Sus atributos son los siguientes:

- `_kf`: Instancia de la clase `KFMatrix`. Contiene los datos de la medida de posición en forma matricial.
- `_time`: Marca de tiempo en la que se tomó la medida, es el criterio de ordenación de las medidas de cada objeto.
- `_id`: Identificador del objeto al cual pertenece la medición.
- `_sensorType`: Identifica el tipo de sensor que ha tomado la medida.
- `_hasVel`: Indica si la medida es solamente de posición o si además tiene datos de la velocidad del objeto.
- `_covXX`, `_covXY`, `_covYY`: Valores de covarianza entre las diferentes medidas.

Esta clase es una clase de almacenamiento de datos por lo que las funciones que implementa son de consulta y de modificación de atributos.

4.4.2.2 CLASE KFQUEUE

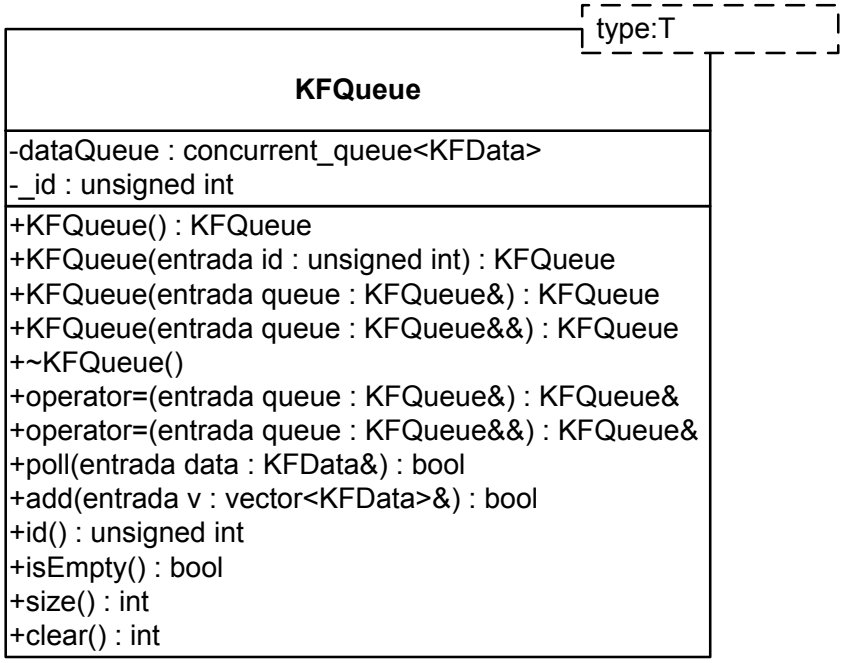

Figura 4.6. Clase *KFQueue*.

La clase `KFQueue` representa las colas que contendrán las medidas, encapsuladas en un objeto `KFData`, que vayan llegando en cada instante. A cada objeto monitorizado por el sistema le es asignado un identificador y una cola, etiquetada con ese mismo identificador en la cual se almacenan sus medidas.

Dado que el algoritmo del filtro de Kalman es un algoritmo iterativo, los datos de medidas deben ser tratados de forma ordenada respecto al instante en que se tomaron, es decir, si existen dos medidas de un mismo objeto, una de ellas tomada en el instante *t* y la otra tomada en el instante *t-1*, se debe garantizar que, en ningún caso se tratará primero la medida tomada en el instante *t* y posteriormente la tomada en el instante *t-1*. Debido a esto los datos deben insertarse de forma ordenada. En la implementación se ha considerado, como ya se ha dicho, que los datos llegan ordenados, por lo que la ordenación se implementa en el simulador de envío de datos que se explicará más adelante.

Esta cola deberá soportar operaciones concurrentes ya que aunque nunca pueda haber dos hilos extrayendo datos de la misma, si se dará la situación en que un hilo extraiga un dato para ser tratado mientras otro inserta nuevos datos.

Los atributos de la clase `KFQueue` son los siguientes:

- `dataQueue`: Es el contenedor donde se almacenan las medidas al objeto asociado a la cola, es un contenedor concurrente.
- `_id`: Identificador que asocia la cola con un objeto móvil.

Las funciones que se implementan en esta clase ofrecen la funcionalidad necesaria para realizar operaciones de inserción y extracción.

4.4.2.3 CLASE KFTASKMANAGER

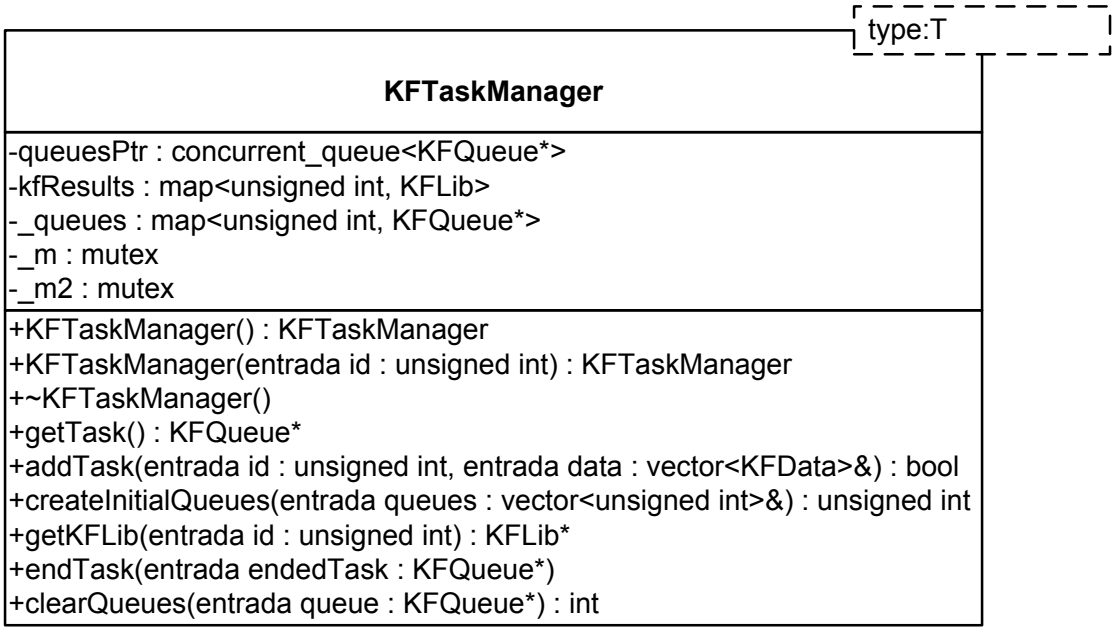

Figura 4.7 Clase KFTaskManager

La clase `KFTaskManager` es la encargada de gestionar las diferentes colas de datos que compondrán el sistema. Esta clase se encarga de mantener referencias a las colas de datos de cada objeto, así como a una colección de instancias `KFLib` donde se almacenan los resultados que se van generando para cada objeto móvil. Los atributos que forma la clase son:

- `queuesPtr`: Contenedor de punteros a objetos `KFQueue`. Alberga puntero a las colas de datos de los objetos móviles. Cuando un hilo solicita una tarea se extrae el puntero que está en la cabeza de la cola, cuando el hilo termina de tratar el dato correspondiente se vuelva a insertar el puntero al final de la cola. De esta manera

se evita que dos hilos puedan obtener la misma cola y tratar datos del mismo objeto móvil de forma desordenada sin tener que realizar bloqueos innecesarios sobre la cola.

- `kfResults`: Contenedor que almacena instancias de la clase `KFLib`. Cada instancia es el estado actual de un objeto móvil. El contenedor es de tipo asociativo y cada objeto queda indexado por el identificador de objeto móvil.
- `_queues`: Contenedor que almacena las colas de datos. Al igual que el `kfResults` es un contenedor asociativo cuyos datos se indexan por el identificador de objeto móvil.
- `_m`, `_m2`: Mutex de la biblioteca estándar de C++. Se utilizan para controlar el acceso a los contenedores `kfResults` y `_queues`, ya que estos contenedores no son concurrentes.

Las funciones de esta clase permiten manejar las colas de datos del sistema, así como gestionar la forma y el orden en que se tratan las tareas.

Finalmente en la Figura 4.8 se muestra la relación entre las distintas clases del sistema.

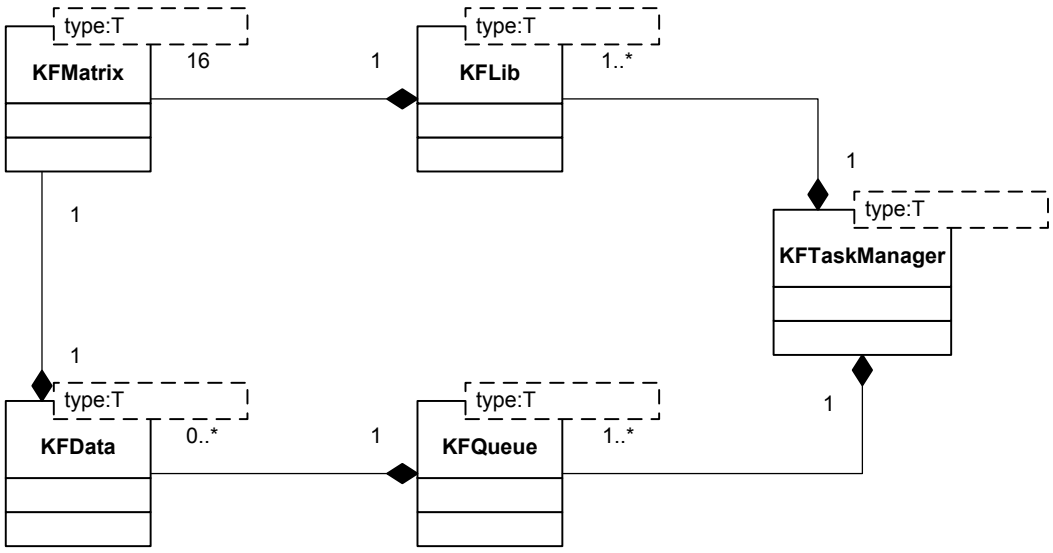

Figura 4.8 Diagrama de clases del sistema completo

4.4.3 CLASE AUXILIAR *DATALOADER*

Debido a que no ha sido posible recibir los datos directamente de los aparatos de medida, sino que se han cargado desde fichero almacenados en disco, se ha implementado una clase, llamada `DataLoader`, que simula la llegada de datos en tiempo real.

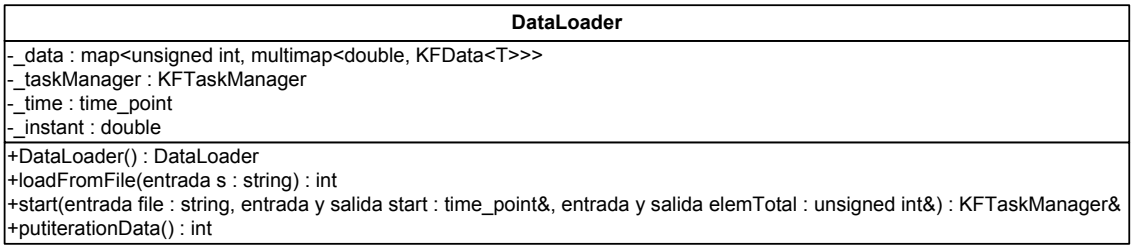

Figura 4.9 Clase *DataLoader*

Los atributos que forman esta clase son los siguientes:

- `_data`: Es un contenedor donde se almacenan todos los datos cargados de fichero, para no tener que acceder a disco constantemente.
- `_taskManager`: Es una instancia de un objeto `KFTaskManager`, se utiliza por razones de rapidez y practicidad a la hora de inicializar el sistema para realizar las pruebas.
- `_time`: objeto de tipo `time_point` que almacena el instante en que se ejecutó la función `putIterationData`.
- `_instant`: variable de tipo `double` que almacena en segundos la última vez que se ejecuto el método `putIterationData`.

La clase `DataLoader` permite cargar los datos que se utilizarán para realizar las pruebas y que se encuentran almacenados en ficheros de texto a través de la función `loadFormFile`. Además permite simular la llegada de datos en instantes determinados e introducirlos en las colas de datos de los objetos móviles a través de la función `putIterationData` cuyo funcionamiento se explica en la Figura 4.10. Por último, dado que esta clase es necesaria para probar todas las implementaciones que se han desarrollado se ha incluido una función llamada `start` que permite inicializar todo lo necesario para ejecutar las pruebas de forma rápida.

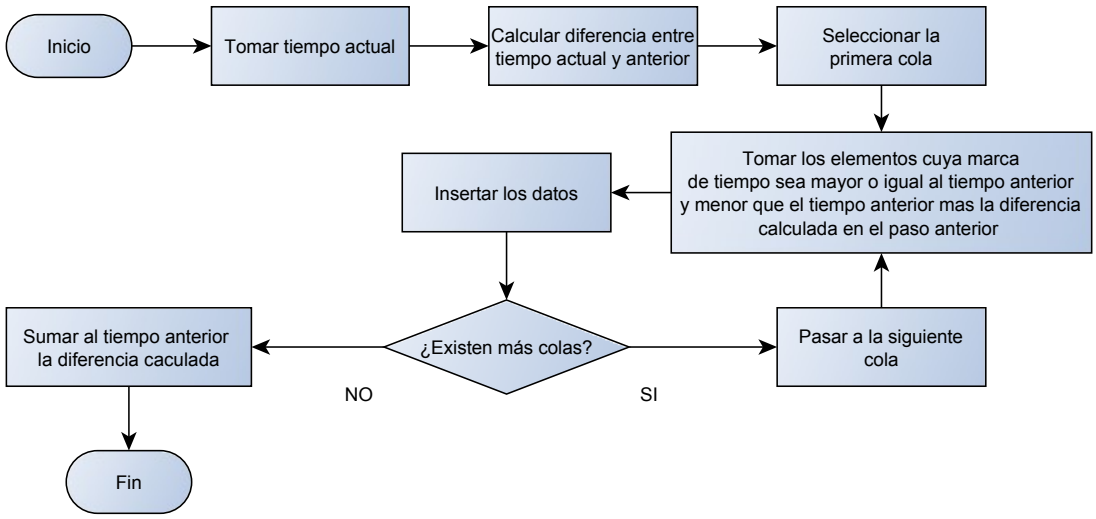

Figura 4.10 Diagrama de flujo de la función *putIterationData*

La Figura 4.10 representa el funcionamiento de la función `putIterationData`. Como ya se ha indicado, esta función simula la llegada de datos en tiempo real. La clase `DataLoader` tiene un atributo que almacena el instante de tiempo en que esta función se ejecutó por última vez, al inicio de la ejecución este atributo vale 0. Cuando la función comienza a ejecutar se toma el tiempo y se calcula cuánto tiempo ha pasado desde la ejecución anterior. Esta diferencia de tiempo se almacena en una variable local.

Posteriormente se entra en un bucle que recorre todos los contenedores donde están almacenados los datos que se han cargado de fichero. Cada uno de estos contenedores se encarga de almacenar los datos de un objeto móvil. Por cada uno de estos contenedores se seleccionan todos los elementos cuya marca de tiempo se encuentra entre el valor de la ejecución anterior de la función y el valor de la ejecución anterior más la diferencia que se ha calculado previamente, es decir los elementos cuya marca de tiempo es menor al instante actual y que no se habían insertado previamente. Estos contenedores mantienen los datos ordenados por marca de tiempo.

Finalmente se insertan los datos seleccionados en la cola de datos correspondiente, esto se consigue llamando a la función `addTask` de la clase `KFTaskManager`. Cuando se hayan recorrido todos los contenedores, y por tanto, insertado las medidas de todos los objetos móviles en su correspondiente cola, se actualiza el atributo que almacena el instante de la ejecución anterior para dejarlo listo para la siguiente y termina la función.

5 DESARROLLO

En este capítulo se detallan las mejoras que se han ido introduciendo desde la versión secuencial del algoritmo hasta el sistema final.

5.1 OPTIMIZACIONES DEL CÓDIGO SECUENCIAL

En este punto se detallan las mejoras que se han introducido en la versión secuencial del sistema para optimizar su funcionamiento antes de introducir paralelismo. Estas optimizaciones se han realizado sobre la clase `KFMatrix` ya que esta clase es la que más capacidad de cómputo consume por ser la que engloba las operaciones algebraicas necesarias para la aplicación del algoritmo.

5.1.1 SEMÁNTICA DE MOVIMIENTO DE C++11

Una de las novedades del estándar C++11 es la semántica de movimiento. Esta semántica evita tener que realizar copias profundas de valores temporales ya que simplemente se mueve el puntero a los datos. De esta forma se ahorra una gran cantidad de tiempo cuando, por ejemplo, se asigna el valor de retorno de una función miembro a una variable.

En general, cuando una función llamada devuelve un valor a la función llamante y esta lo asigna a una variable se realizan dos copias en memoria de los datos devueltos, la primera de ellas, desde el marco de pila de la función llamada al marco de pila de la función llamante y la segunda, desde el marco de pila de la función llamante al espacio de memoria de la variable destino (que habitualmente estará en el propio marco, aunque puede estar en el *heap*).

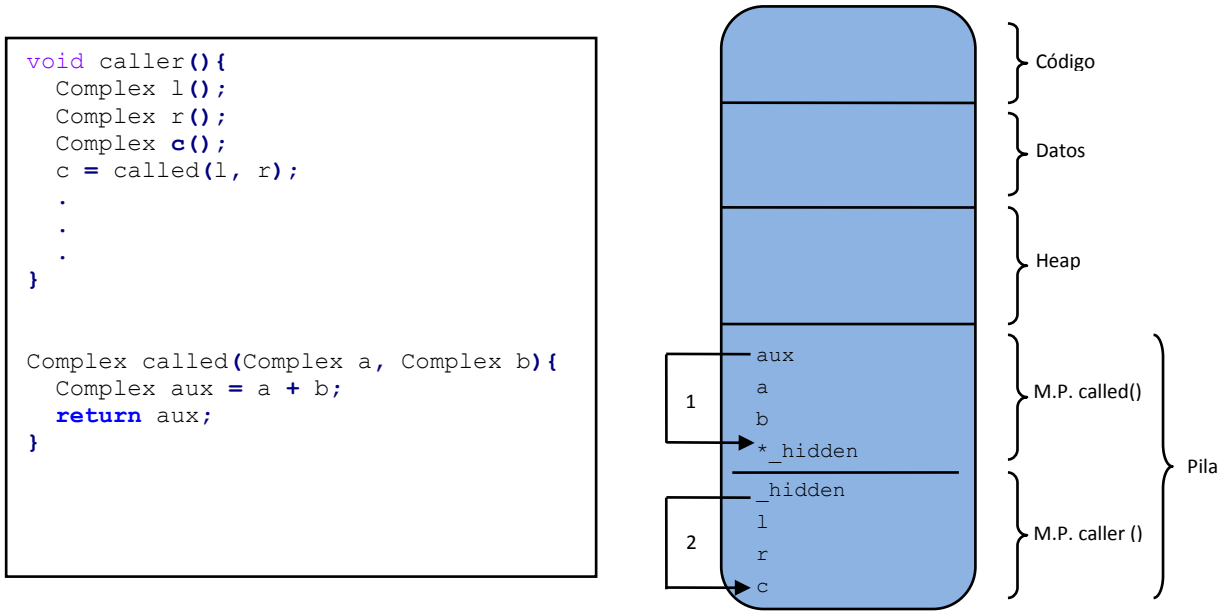

Figura 5.1 Movimientos de datos en una llamada a función.

Como se puede observar en la Figura 5.1, sin ninguna optimización deben hacerse dos copias del resultado devuelto por una función, una desde el marco de pila de la función llamada al marco de pila de la función llamante y otra a la variable destino. La forma de realizar esto en la mayoría de los compiladores es añadir código a las funciones de manera que a la función llamada se le pasa un puntero a un valor creado en la función llamante para que almacene en él el resultado y finalmente se copia este valor a la variable destino[18]. El código generado por el compilador sería equivalente al de la Figura 5.2.

```
void caller(){
  Complex l();
  Complex r();
  Complex c();
  Complex _hidden();
  c = *called(l, r, &_hidden);

  .
  .
  .
}


Complex* called(Complex a, Complex b, Complex* _hidden){
  Complex aux = a + b;
  *_hidden = aux;
  return _hidden;
}
```

Figura 5.2 Código equivalente generado en una llamada a función

Para evitar estas copias la mayoría de compiladores modernos utilizan la optimización del valor de retorno (RVO, por sus siglas en inglés *Return Value Optimization*). La RVO se basa en evitar, dependiendo de la implementación, una o las dos copias de datos. Para ello el compilador suele aprovechar la variable oculta que crea al generar el código, copiando en ella el resultado que se va a retornar evitando así la primera de las copias o, directamente pasando como parámetro a la función llamada una referencia a la variable destino, evitando de esta manera ambas copias. En el primer caso se tendría un código similar al de la Figura 5.3, la Figura 5.4 representa el segundo caso.

```
void caller(){
  Complex l();
  Complex r();
  Complex c();
  Complex _hidden();
  c = *called(l, r, &_hidden);

  .
  .
  .
}


Complex* called(Complex a, Complex b, Complex* _hidden){
  *_hidden = a + b;
  return _hidden;
}
```

Figura 5.3. Código equivalente en una llamada a función con RVO evitando una copia.

```
void caller(){
  Complex l();
  Complex r();
  Complex c();
  called(l, r, &c);
  .
  .
  .
}


Complex* called(Complex a, Complex b, Complex* c){
  *c = a + b;
  return;
}
```

Figura 5.4. Código equivalente en una llamada a función con RVO evitando ambas copias.

Sin embargo no todos los compiladores ni en todas las situaciones realizan esta optimización, por lo que en ocasiones se realizan ambas copias o al menos una de ellas, para subsanar este inconveniente se ha utilizado la semántica de movimiento de C++11.

En el estándar C++03 y anteriores los *rvalues* temporales fueron pensados para no poder ser modificados, en el nuevo estándar C++11 se introduce un nuevo tipo de referencia no constante a la que se denomina referencia *rvalue* y la cual se indica mediante *&&* que permite que los temporales sean modificados después de haberse inicializado. Esto permite que en un constructor de copia que toma como parámetro una referencia r*value* se pueda apuntar el nuevo objeto a los datos apuntados por el temporal y después fijar el puntero del *rvalue* temporal a `null` evitando así la copia profunda. Por otra parte al poder haberse fijado el puntero del temporal a `null` su memoria no es borrada cuando termina la función y queda fuera de ámbito, siendo esta zona de memoria referenciada por el nuevo objeto.

```
Complex::Complex(Complex&& C){

    //Movimiento de datos.
    this->_data = C._data;
    this->_length = C._length;

    //Eliminación del temporal.
    C._data = nullptr;
    C._length = 0;
}
```

Figura 5.5 Constructor de movimiento de una clase compleja.

Para una clase compleja formada por un array y un entero que indica su longitud se tendría un constructor de movimiento como el de la Figura 5.5. Se puede observar como en lugar de copiarse todos los datos del array `_data` simplemente se copia la dirección de comienzo del mismo, evitando copiar dato por dato, lo que produce accesos a memoria mínimos. Al terminar la función se invoca al destructor de la clase para el objeto temporal `C` ya que su ámbito es el propio constructor pero los datos del array no se eliminan ya que `_data` apunta a `null` y `_length` se ha copiado por valor por lo que no hay inconveniente en liberar su memoria.

```
int main()
{
    std::string str = "Hello";
    std::vector<std::string> v;

    //Copia de la cadena dentro del vector
    v.push_back(str);
    std::cout << "Tras la copia, str: \"" << str <<"\"\n";

    //Utilizando un rvalue reference se sustituye la
    //copia profunda por un movimiento de los datos.
    //str quedará vacio.
    v.push_back(std::move(str));
    std::cout << "Tras el movimiento, str: \"" << str << "\"\n";

    std::cout << "El contenido del vector es: \"" << v[0]
                                        << "\", \"" << v[1] << "\"\n";
}
```

Figura 5.6 Utilización de la función std::move()

Hay que tener en cuenta que la semántica de movimiento implica que el objeto a copiar se destruye tras la copia por lo que en el caso en que el parámetro del constructor sea una variable con nombre el compilador nunca generará código que invoque al constructor de movimiento ya que la variable puede ser utilizada posteriormente. En los casos en que esto ocurra es el programador el que explícitamente debe invocar el constructor de movimiento, para ello se utiliza la plantilla de función `std::move<T>()`. Al invocar esta función pasándole como parámetro el objeto a copiar se fuerza a que se invoque al constructor de movimiento. El resultado de ejecutar el código de la Figura 5.6 es el siguiente:

```
"Tras la copia, str: "Hello"
"Tras el movimiento, str: ""
"El contenido del vector es: "Hello", "Hello"
```

Como se puede apreciar tras ejecutar la función `move` la cadena `str` queda vacía debido a que ahora apunta a `null` y su dirección anterior es ahora apuntada por la posición 1 del vector.

Aplicado al algoritmo de los filtros de Kalman la semántica de movimiento produce una gran mejora ya que se trabaja con matrices. Estas matrices se implementan con vectores de valores en coma flotante y pueden llegar a ser bastante grandes por lo que la copia de un objeto puede llegar a ser una operación muy costosa. Para evitar esto en la clase `KFMatrix`, que implementa las matrices que se utilizan para el ejecutar el algoritmo, se han definido dos funciones que implementan semántica de movimiento, un constructor copia con semántica de

movimiento, es decir, un constructor de movimiento y un operador de asignación con semántica de movimiento, de esta forma se pueden hacer copias con semántica de movimiento tanto en objetos sin inicializar previamente, utilizando el constructor, como en objetos previamente inicializados utilizando el operador de asignación sobrescrito. Hay que tener en cuenta que si la variable destino ha sido previamente inicializada debemos liberar sus recursos antes de cambiar sus punteros ya que si no perderíamos la referencia a sus recursos y no podríamos liberar esa memoria produciendo un *leak.*

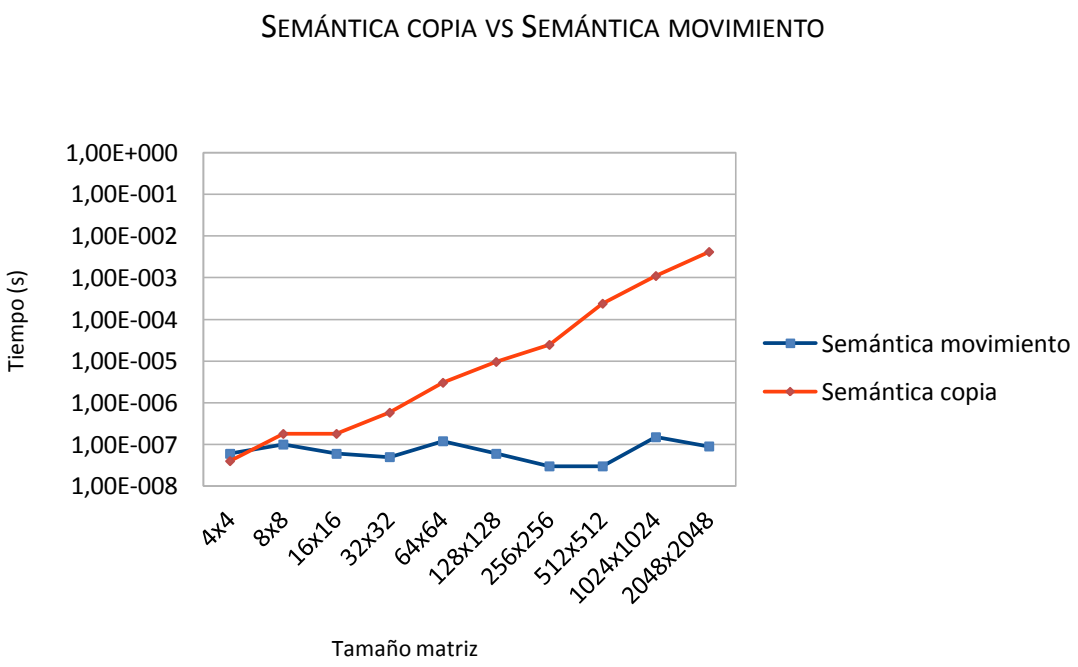

Figura 5.7 Tiempo de asignación para semántica de copia y de movimiento

La Figura 5.7 ilustra el tiempo que se tarda en realizar una asignación para matrices de distintos tamaños compuestas por elementos de tipo `double`. Como se puede apreciar el coste usando la semántica de movimiento es constante respecto de la cantidad de datos, sin embargo, en el caso de la semántica de copia el tiempo crece exponencialmente con el tamaño de los datos a copiar llegando ser hasta 5 órdenes de magnitud mayor en el peor caso.

5.1.2 USO DE CONTENEDORES ESTÁNDAR DE C++

El uso de matrices en el algoritmo hace necesaria la utilización de algún tipo de estructura para almacenarlas. En el código de partida el almacenamiento se hacía en *arrays* estilo C. El uso de *arrays* de C en C++ no es recomendable debido a que son estructuras de bajo nivel que pueden dar lugar a numerosos fallos de programación, sobre todo de gestión de memoria, y a que hoy en día existen estructuras que ofrecen el mismo rendimiento que los *arrays* y evitan muchos de sus problemas.

En C++ se pueden tener dos tipos de *array* según su inicialización, estáticos y dinámicos. Los *arrays* estáticos son los inicializados en la pila de la función en la que se encuentra su declaración, su ámbito es el de la propia función y se libera su memoria cuando esta acaba, por lo que en este caso no es necesario hacer una liberación explicita de ella, estos *arrays* tienen un tamaño fijo que no se puede modificar. Los *arrays* dinámicos son aquellos que se inicializan con el operador *new*, el cual reserva memoria dinámica en el *heap* del proceso y por tanto permanecerán en memoria hasta el final del proceso o hasta que esta memoria reservada se libere utilizando el operador *delete[]* el cual llamará a todos los destructores de los elementos del *array*. Estos *arrays* pueden modificar su tamaño en tiempo de ejecución reservando más memoria dinámica. El gran problema de estos *arrays* dinámicos es que en ocasiones se pueden producir pérdidas o *leaks* de memoria ya que es el propio programador quién manualmente debe liberarla.

Por otra parte cuando se pasa un *array* como argumento a una función, este se pasa como un puntero a la primera posición, por lo que se pierde cualquier información de tamaño en el mismo, obligando a pasar otro argumento que indique el tamaño.

C++ contiene una biblioteca llamada *Standard Tamplate Library* o *STL* que contiene una gran cantidad de contenedores de datos, así como iteradores para recorrerlos de forma eficiente, además provee de los algoritmos más comunes en informática. Esta librería es genérica con lo que sus clases están altamente parametrizadas permitiendo usarlas con cualquier tipo de datos, ya sea básico o definido por el usuario.

Uno de los contenedores ofrecidos por esta biblioteca es std::*vector*. Este contenedor es equivalente a un *array* capaz de crecer de forma dinámica, sin embargo encapsula la gestión de memoria por parte del programador evitando así los errores de gestión de memoria. Por otra parte *std::vector* ofrece una función miembro llamada *size()* que permite saber en tiempo

de ejecución el tamaño de un vector, evitando el problema de tener que pasarlo por parámetro.

Otra de las ventajas de las clases que componen la biblioteca es el uso de iteradores para recorrer los contenedores. Los iteradores son objetos que hacen la misma función que los punteros, sin embargo en el caso de los iteradores la biblioteca asegura que para cada contenedor su iterador lo recorrerá de la forma más óptima posible. Además de con iteradores o punteros, la clase *std::vector* permite el acceso a sus elementos a través del operador *[]* por lo que se puede utilizar exactamente igual que un *array* estilo C.

A priori puede parecer que el uso de *std::vector* conlleva una sobrecarga adicional ya que es una clase compleja que soporta varias operaciones y guarda información además del contenido del propio vector, como su tamaño o iteradores. Para comprobar esto se han realizado dos pruebas distintas, la primera de ellas ha sido medir el tiempo que tardan las operaciones, suma, producto, inversa y asignación de matrices utlizando una implementación con arrays y otra con *std::vector*. La siguiente prueba ha sido analizar, para un código muy sencillo, cual es el código en ensamblador que genera el compilador al trabajar con iteradores y con punteros convencionales sobre un vector.

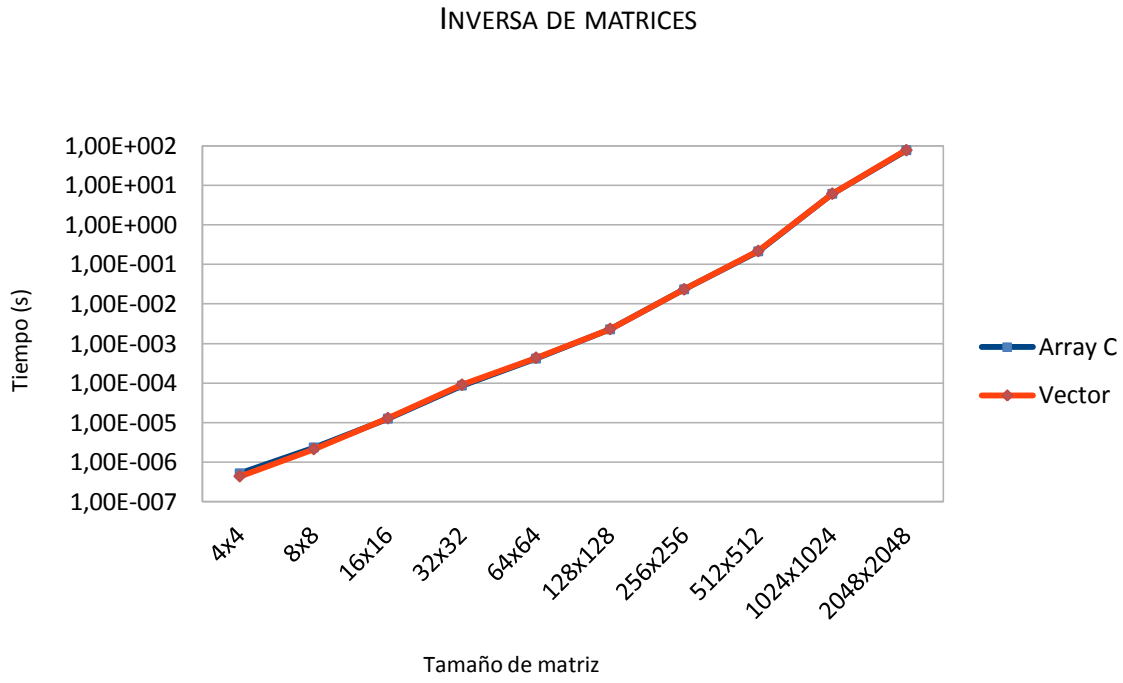

Figura 5.8 Tiempo empleado en una inversión de matrices por un array y un vector

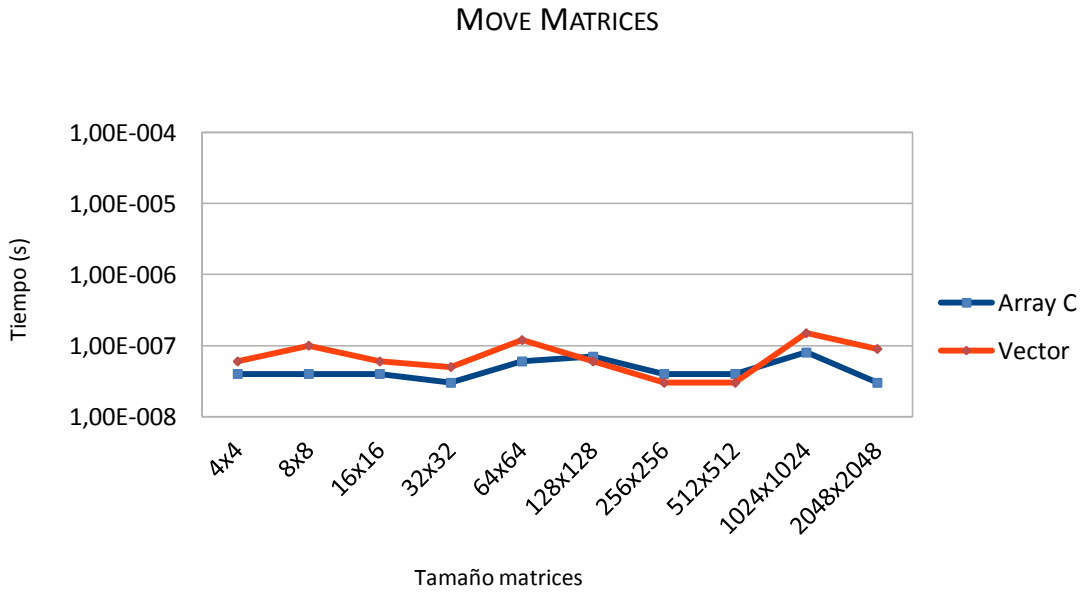

Figura 5.9 Tiempo empleado en una asignación con movimiento para un array y un vector

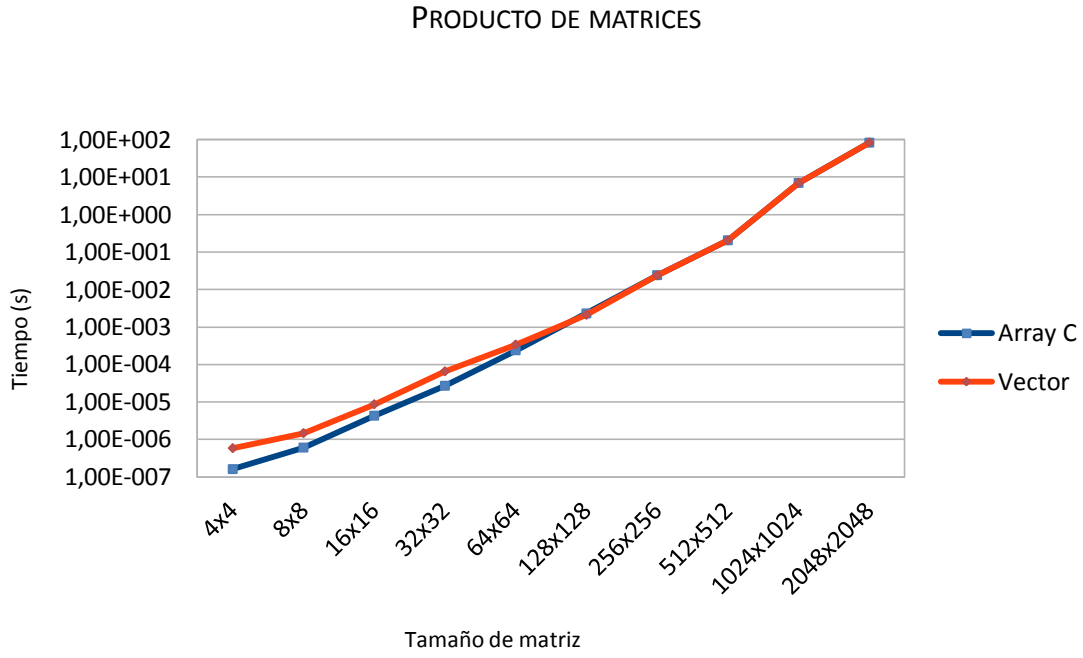

Figura 5.10 Tiempo empleado en un producto de matrices para un array y un vector

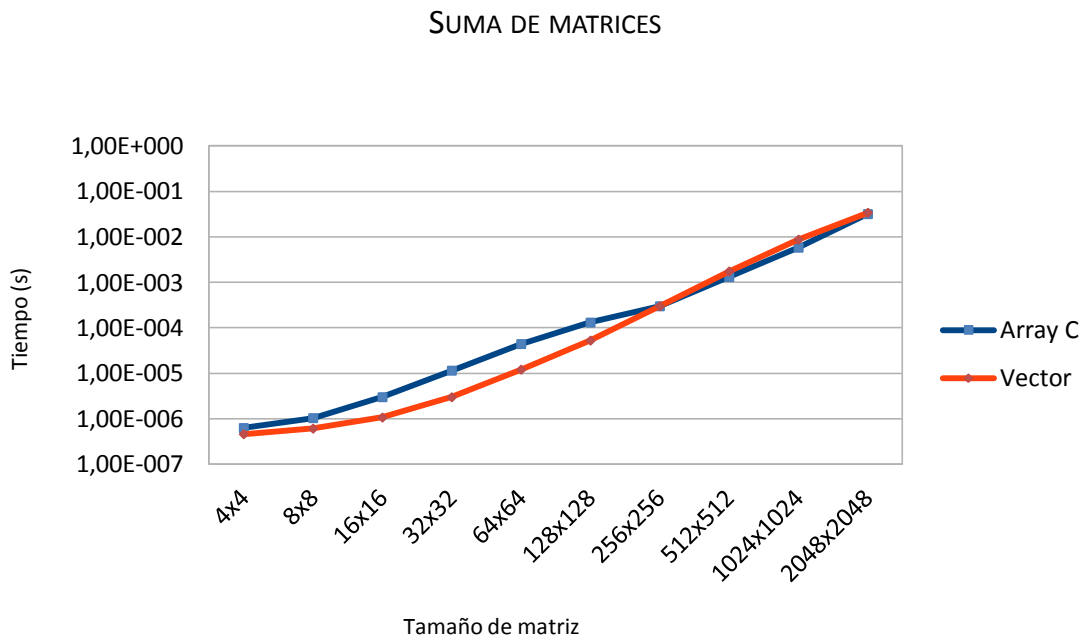

Figura 5.11 Tiempo empleado en una suma de matrices para un array y un vector

Como se puede observar los tiempos son muy similares para las implementaciones que utilizan *std::vector* y las que utilizan *arrays* de C. La razón de ello es que internamente ambos son muy similares, ya que tanto *std::vector* como los *array* son almacenados de forma contigua en memoria por lo que su acceso es igual de costoso.

Por otra parte los resultados parecen indicar que tanto los punteros como los iteradores son internamente lo mismo, para comprobar esto se ha realizado una prueba con el código de la Figura 5.12, en la que se observa que para operaciones equivalentes de *arrays* y vectores y operaciones equivalentes de iteradores y punteros el código generado en ensamblador es el mismo. En conclusión el tiempo que se tarda en acceder a un dato almacenado en un *std::vector* y en un *array* es el mismo, sin embargo los objetos de la clase *std::vector* proporcionan mecanismos que facilitan enormemente la programación y que evitan en gran medida errores que con los *arrays* son muy comunes.

```
#include <vector>
struct S
{
  std::vector<int> v;
  int * p;
  std::vector<int>::iterator i;
};
//Indexacion de un puntero
int index_puntero (S & s) { return s.p[3]; }
    //movq  24(%rdi), %rax
    //movl  12(%rax), %eax
    //ret

//Indexacion de un vector
int index_vector (S & s) { return s.v[3]; }
    //movq  (%rdi), %rax
    //movl  12(%rax), %eax
    //ret

//Desreferencia de un puntero
int desref_puntero (S & s) { return *s.p; }
    //movq  24(%rdi), %rax
    //movl  (%rax), %eax
    //ret

 //Desreferencia de un iterador
int desref_iterador (S & s) { return *s.i; }
    //movq  32(%rdi), %rax
    //movl  (%rax), %eax
    //ret

//Incremento de un puntero
void inc_puntero (S & s) { ++s.p; }
    //addq  $4, 24(%rdi)
    //ret

//Incremento de un iterador
void inc_iterador (S & s) { ++s.i; }
    //addq  $4, 32(%rdi)
    //ret
```

Figura 5.12 Código C++ y su correspondiente ensamblador generado.[1]

[1] Compilación: g++ -O3 -S *<nombre_del_fichero.cpp>*. Con GCC 4.6.1 en Ubuntu 12.04 LTS

5.2 MEDICIONES DEL TIEMPO DE EJECUCIÓN CON *KCACHEGRIND*

El primer paso antes de paralelizar ha sido buscar las funciones del algoritmo que más tiempo consumían, para ello se ha utilizado la herramienta de profiling KCacheGrind, la cual permite medir, entre otras cosas, el tiempo que tardan en ejecutarse las funciones de un código.

Para llevar a cabo la medición se ha ejecutado una prueba que calcula 100 iteraciones del algoritmo del filtro de Kalman sobre mediciones de 4 estados.

KCacheGrind necesita de la ejecución previa de Valgrind, como ya se ha explicado en el punto 2.9.1 de este documento. Ambos están disponibles como paquete en los repositorios oficiales de Ubuntu 12.04LTS, el cual es el sistema operativo sobre el que se ha desarrollado y ejecutado el sistema.

Antes de poder ejecutar Valgrind y su herramienta Callgrind es necesario haber compilado el código con símbolos de depuración para que posteriormente KCacheGrind pueda mostrar la información asociada a cada función del código.

Para medir los tiempos con KCacheGrind es necesario ejecutar el sistema con Valgrind y su herramienta Callgrind, el comando de ejecución que se utiliza es el siguiente:

```
valgrind --tool=callgrind ./<programa a ejecutar>
```

La ejecución de este comando produce un fichero de salida cuyo nombre será *cachegrind.out.<PID del programa que se ejecutó>.* Posteriormente se ejecuta KCacheGrind y se abre el fichero anterior, obteniendo una pantalla como la mostrada en la Figura 5.13.

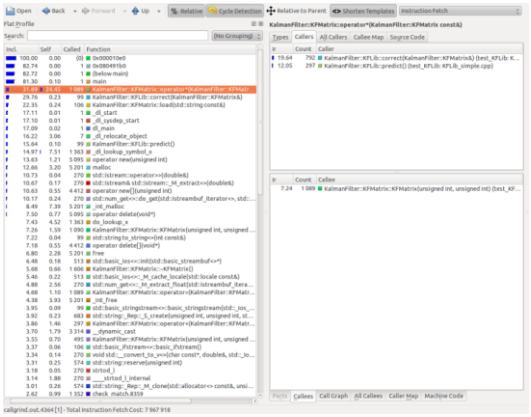

Figura 5.13. Pantalla inicial de *KCachegrind.*

Esta pantalla se compone de tres partes. En la parte izquierda se muestra una lista de funciones que se han ejecutado y su coste porcentual de tiempo respecto al total del programa. En la Figura 5.14 se muestra esta parte de la pantalla.

Incl.	Self	Called	Function
100.00	0.00	(0)	0x000010e0
82.74	0.00	1	0x080491b0
82.72	0.00	1	(below main)
81.30	0.10	1	main
31.69	24.45	1 089	KalmanFilter::KFMatrix::operator*(KalmanFilter::KFMatr…
29.76	0.23	99	KalmanFilter::KFLib::correct(KalmanFilter::KFMatrix&)
22.35	0.24	106	KalmanFilter::KFMatrix::load(std::string const&)
17.11	0.01	1	_dl_start
17.10	0.01	1	_dl_sysdep_start
17.09	0.02	1	dl_main
16.22	3.06	7	_dl_relocate_object
15.64	0.10	99	KalmanFilter::KFLib::predict()
14.97	7.51	1 363	_dl_lookup_symbol_x
13.63	1.21	5 095	operator new(unsigned int)
12.66	3.20	5 201	malloc
10.73	0.04	270	std::istream::operator>>(double&)
10.67	0.17	270	std::istream& std::istream::_M_extract<>(double&)
10.63	0.55	4 412	operator new[](unsigned int)
10.17	0.24	270	std::num_get<>::do_get(std::istreambuf_iterator<>, std::…
8.49	7.39	5 201	_int_malloc
7.50	0.77	5 095	operator delete(void*)
7.43	4.52	1 363	do_lookup_x
7.26	1.59	1 090	KalmanFilter::KFMatrix::KFMatrix(unsigned int, unsigned …
7.22	0.04	99	std::string to_string<>(int const&)

Figura 5.14. Detalle de parte izquierda de la pantalla de *KCachegrind.*

La lista de funciones la componen 4 columnas. La primera de ellas es el porcentaje de tiempo respecto del total que tarda cada función en ejecutar, incluyendo las subrutinas a las que invoca, este coste se conoce como coste inclusivo. La segunda columna es el coste exclusivo, es decir, el porcentaje de tiempo respecto del total que tarda la función sin tener en cuenta las subrutinas a las que invoca. La tercera columna indica el número de veces que se ha invocado la función en cuestión. Por último la cuarta columna muestra el prototipo de la función en cuestión. Como se puede observar la función cuyo coste es mayor, sin contar `main`, es el operador de producto, que consume alrededor de un tercio del total de tiempo, por lo que la reducción del tiempo de esta operación producirá una reducción importante del tiempo global.

En la parte derecha de la pantalla de inicio de KCachegrind se muestran las funciones que invocan y que son invocadas por la función que hayamos seleccionado en la lista de la parte izquierda.

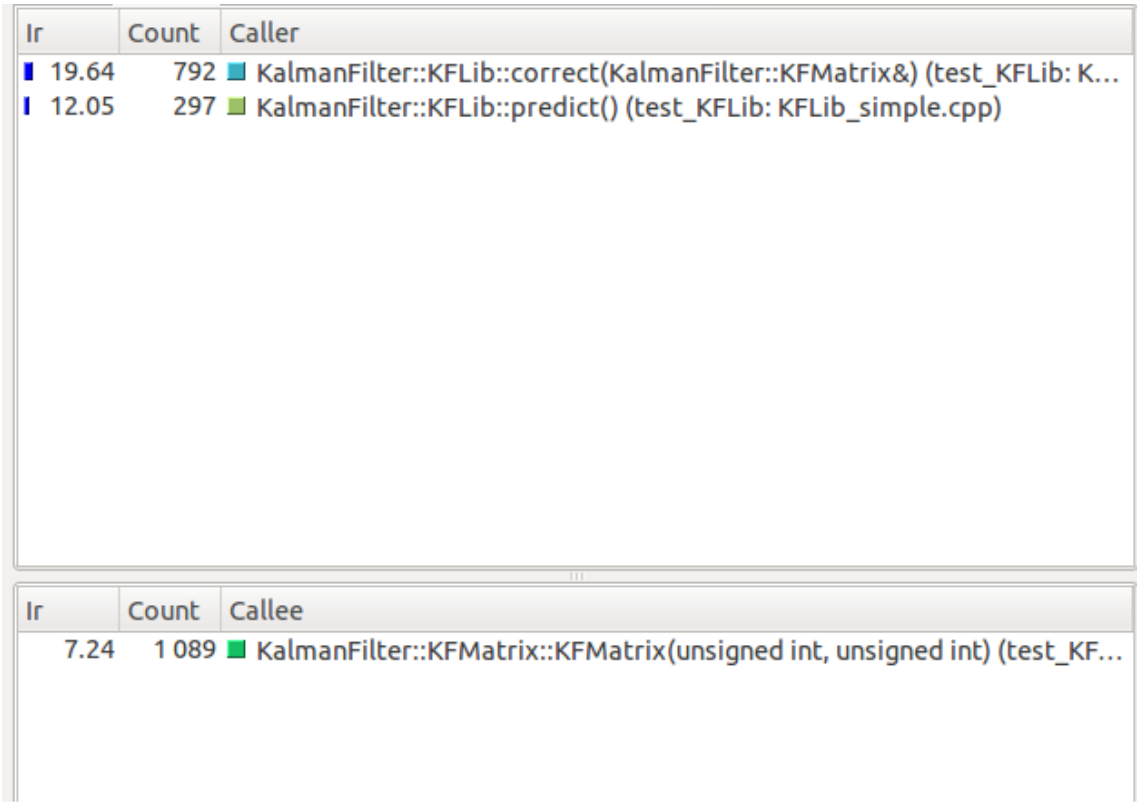

Figura 5.15. Parte derecha de la pantalla inicial de *KCachegrind*.

En la parte superior de la Figura 5.15 se muestran las funciones que han invocado al operador producto. Esta vista se compone de filas divididas en 3 columnas. Cada fila representa una función que ha llamado, en este caso, al operador producto. En cada fila la primera columna muestra el coste respecto del total que ha tenido para la función en cuestión invocar al operador producto. La segunda muestra las veces que se ha invocado el operador producto por la función y la tercera muestra el prototipo de la función en cuestión. La suma de los costes de todas las funciones que invocan al operador producto es el coste inclusivo del propio operador producto.

En la parte inferior las funciones que han sido invocadas por el operador producto. La vista es muy similar a la anterior. En la primera columna se muestra el porcentaje de tiempo que el operador producto ha consumido llamando a la función. En la segunda se muestran las veces que ha llamado a la función y en la tercera el prototipo de la función. En este caso se puede observar que restamos el coste de todas las funciones invocadas por el operador producto a su coste inclusivo obtenemos su coste exclusivo.

La siguiente figura muestra el coste de todas las funciones implementadas para la aplicación del algoritmo de los filtros de Kalman y su coste.

ncl.	Self	Called	Function
31.69	24.45	1 089	KalmanFilter::KFMatrix::operator*(KalmanFilter::KFMatrix const&)
29.76	0.23	99	KalmanFilter::KFLib::correct(KalmanFilter::KFMatrix&)
22.35	0.24	106	KalmanFilter::KFMatrix::load(std::string const&)
15.64	0.10	99	KalmanFilter::KFLib::predict()
7.26	1.59	1 090	KalmanFilter::KFMatrix::KFMatrix(unsigned int, unsigned int)
5.68	0.66	1 606	KalmanFilter::KFMatrix::~KFMatrix()
4.68	1.10	1 089	KalmanFilter::KFMatrix::operator=(KalmanFilter::KFMatrix const&)
3.86	1.46	297	KalmanFilter::KFMatrix::operator+(KalmanFilter::KFMatrix const&)
3.55	0.70	495	KalmanFilter::KFMatrix::KFMatrix(unsigned int, unsigned int, double co…
2.22	1.07	198	KalmanFilter::KFMatrix::operator-(KalmanFilter::KFMatrix const&)
0.95	0.95	99	KalmanFilter::KFMatrix::inverse()
0.85	0.16	106	KalmanFilter::KFMatrix::copy(unsigned int, unsigned int, double const*)
0.12	0.00	1	KalmanFilter::KFLib::KFLib(KalmanFilter::KFMatrix&, KalmanFilter::KF…
0.07	0.02	10	KalmanFilter::KFMatrix::KFMatrix(KalmanFilter::KFMatrix const&)
0.05	0.00	1	KalmanFilter::KFLib::~KFLib()
0.02	0.01	2	KalmanFilter::KFMatrix::transpose()
0.01	0.01	198	KalmanFilter::KFLib::getP()
0.01	0.01	198	KalmanFilter::KFLib::getX()
0.01	0.01	1	KalmanFilter::KFMatrix::identity()
0.00	0.00	1	_GLOBAL__sub_I__ZN12KalmanFilter5KFLibC2Ev
0.00	0.00	1	_GLOBAL__sub_I__ZN12KalmanFilter8KFMatrixC2Ev
0.00	0.00	11	KalmanFilter::KFMatrix::KFMatrix()
0.00	0.00	2	KalmanFilter::KFMatrix::nrows() const

Figura 5.16. Coste de las funciones implementadas en *KFMatrix* y *KFLib.*

Como se puede observar el mayor coste exclusivo es del operador de producto, en el resto de funciones este coste es muy bajo. Más concretamente en las funciones algebraicas, que son las que se podrían paralelizar, la siguiente función con más coste es el operador de suma, cuyo coste exclusivo no llega al 1,5% del total. Por tanto no merece la pena paralelizar esta función, ni el resto con menos coste, ya que la ganancia sería mínima.

Otra de las vistas que podemos obtener es el mapa de llamadas a funciones del código y el coste de cada una de ellas, partiendo desde la función que deseemos. Este mapa es muy útil para ver de forma rápida el coste que tiene respecto de la función que seleccionemos como raíz las subrutinas a las que invoca. La Figura 5.17 muestra el mapa de llamadas a función desde la función main.

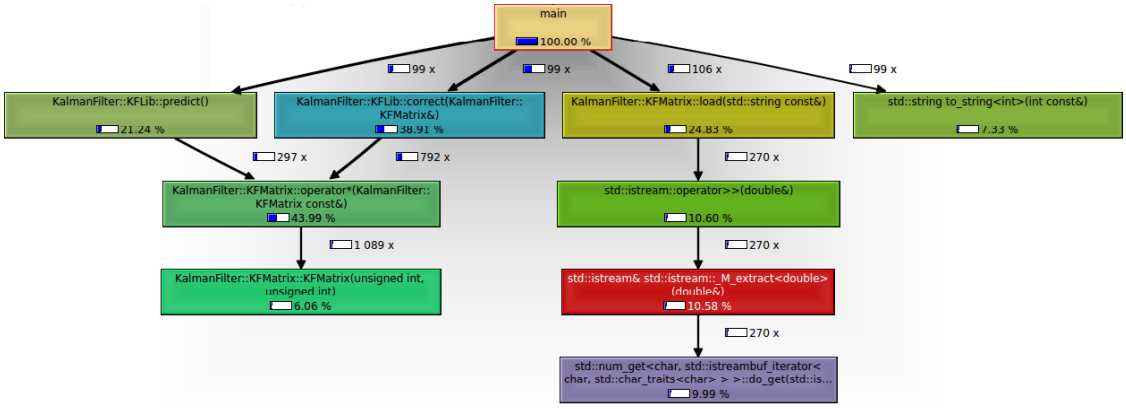

Figura 5.17. Mapa reducido de llamadas a función desde *main*.

El mapa de la Figura 5.17 muestra el árbol de llamadas desde la función main. Por simplicidad *KCachegrind* omite las funciones cuyo coste respecto de la función representada en el nodo raíz es muy pequeño. Como se puede observar los porcentajes de coste en tiempo que se muestran en cada nodo son en referencia al coste de tiempo de la función *main*.

En el mapa se muestra como el operador producto consume prácticamente el 44% del tiempo de ejecución de la función *main*. Este gran porcentaje hace que sea muy recomendable intentar paralelizar esta función como ya se ha dicho anteriormente.

Por último, ahora que hemos visto que funciones es recomendable paralelizar y cuáles no, *KCachegrind* nos permite medir y ver qué porcentaje de tiempo consume cada línea de código respecto al total de la función en que se encuentra.

En la siguiente figura se muestra el porcentaje de tiempo que toma cada línea de la función que implementa el operador producto.

```
...          //! Mul KFMatrix::operator.
...    0.26  KFMatrix KFMatrix::operator*(const KFMatrix& M){
...    0.26    if (this->_ncols != M._nrows){  // sanity check
...              cerr << "Error: operator* error" <<endl;
...              throw;
...            }
...    0.35    KFMatrix result(this->_nrows, M._ncols);
       22.51    1089 call(s) to 'KalmanFilter::KFMatrix::KFMatrix(unsigned int, unsigned int)' (test_KFLib: KFMatrix_simple.cpp)
...    1.39    for (unsigned int i = 0; i < this->_nrows; i++) {
...    4.00      for (unsigned int j = 0; j < M._ncols; j++) {
...   12.90        for (unsigned int k = 0; k < this->_ncols; k++) {
...   58.16          result._matrix[i][j] += this->_matrix[i][k] * M._matrix[k][j];
...                }
...              }
...            }
...    0.04    return result;
...    0.13  }
```

Figura 5.18. Costes del operador producto.

La Figura 5.18 muestra cuales son las partes de código de la función que más tiempo consumen. Lo más costoso es realizar el producto de los elementos de las matrices cuya participación en el tiempo total es cercana al 60%. Tras el producto vemos que la siguiente operación más costosa es crear la matriz resultado, sin embargo esta operación no se puede paralelizar ya que es una llamada a un constructor.

Si sumamos los porcentajes de tiempo que se necesita para realizar el producto, es decir, los 3 bucles y la operación de producto y asignación final tenemos un porcentaje de consumo de tiempo del 76,45% sobre el total de la función. Teniendo que en cuenta que esta función toma el 43,99% del tiempo total del programa tenemos que este código toma un 43,99%∗76,45%=33,63% del tiempo total de ejecución del programa. Este código son una serie de bucles anidados que realizan operaciones con números de coma flotante, por lo tanto es código paralelizable y debido al alto porcentaje de tiempo que lleva ejecutar este fragmento de código se podría lograr una gran mejora global paralelizándolo.

Para el resto de funciones algebraicas no se llevará a cabo ninguna paralelización ya que como se ha visto el porcentaje de tiempo que representa es muy bajo y por tanto la ganancia que se podría obtener también. Las demás funciones, que no realizan operaciones algebraicas, no es posible paralelizarlas ya que son constructores o funciones auxiliares para llevar a cabo entrada/salida.

Cabe destacar que paralelizando el código anterior solo habrá mejora en el porcentaje de tiempo que toma el código que se paraleliza, tal como anuncia la ley de Amdahl, la cual establece que *"la mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que utiliza dicho componente*"[5]. La aceleración o ganancia según Amdahl se calcula como:

$$A = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}} \qquad [5.1]$$

Donde:

1. A es la aceleración o ganancia en velocidad conseguida en el sistema completo debido a la mejora de uno de sus subsistemas.
2. F_m es la fracción de tiempo que el sistema utiliza el subsistema mejorado.
3. A_m es el factor de mejora que se ha introducido en el subsistema mejorado.

5.3 PARALELIZACIÓN CON OPENMP

La primera opción de paralelización que se ha utilizado ha sido OpenMP, como ya se ha descrito OpenMP es una biblioteca de paralelización para C/C++ y Fortran. Una de las ventajas a destacar de OpenMP es que apenas es necesario hacer cambios en el código para introducir paralelismo, por lo que es muy recomendable para paralelizar códigos que ya están escritos previamente de forma secuencial.

Tras la comprobación en el paso anterior de que el producto es la operación que más tiempo consume se ha llevado a cabo la paralelización con OpenMp de esta función. El código de esta función paralelizado es el mostrado en la Figura 5.19.

```
template <class T> KFMatrix<T> KFMatrix<T>::operator*(const KFMatrix<T>& M){
    if (this->_ncols != M._nrows){
        cerr << "Error: operator* error" <<endl;
        throw;
    }
    KFMatrix result(this->_nrows, M._ncols);
    unsigned int total, valor;
    unsigned int nrowsa = this->_nrows;
    unsigned int ncolsa = this->_ncols;
    unsigned int ncolsb = M._ncols;
    unsigned int i,j,k;
    #ifdef _OPENMP
        int nprocs = omp_get_num_procs();
    #endif
    #pragma omp parallel num_threads(nprocs)
    {
        #ifdef _OPENMP
             omp_set_num_threads(nrowsa);
        #endif
        #pragma omp for private(i)
        for (i = 0; i < nrowsa; i++) {
            #pragma omp parallel
            {
                #ifdef _OPENMP
                    omp_set_num_threads(ncolsb);
                #endif
                #pragma omp for private(j,k,valor,total)
                for (j = 0; j < ncolsb; j++) {
                    total = 0;
                    for (k = 0; k < ncolsa; k++) {
                        valor = *(this->_matrix[i]+k) *
                        (*(M._matrix[k]+j));
                        total+=valor;
                    }
                    *(result._matrix[i]+j) = total;
                }
            }
        }
    }
    return move(result);
}
```

Figura 5.19. Código de la función de producto de matrices paralelizado con OpenMP

La función anterior multiplica 2 matrices $(A * B)$, de forma paralela, haciendo uso de la biblioteca OpenMp. La función sobrescribe el operador "*" o producto para la clase `KFMatrix`. Recibe como parámetro un objeto de tipo `KFMatrix` que es la segunda matriz del producto y retorna un objeto `KFMatrix` que contiene la matriz resultado del producto. Para retornar el resultado se utiliza semántica de movimiento, ahorrando así operaciones de memoria.

Primeramente lleva a cabo una verificación del tamaño de las matrices para comprobar que es posible multiplicarlas. A continuación crea las variables locales necesarias para llevar a cabo el producto, como son los índices de los bucles, el objeto `KFMatrix` donde se almacenará el resultado y variables locales para almacenar el tamaño de las matrices.

Tras crear las variables anteriores se ejecuta la función de la biblioteca OpenMP `omp_get_num_procs()`. Esta función devuelve un valor entero que indica el número de procesadores en la máquina disponibles para ejecutar hilos. La función, al igual que todas las funciones de OpenMP que se utilizarán está encerrada en la directiva `#ifdef _OPENMP`, esto permite compilar el código tanto haciendo uso de OpenMp como sin hacer uso de él.

El primer *pragma*, `#pragma omp parallel num_threads(nprocs)`, indica que comienza una zona paralela en la cual el número de hilos que se pueden crear viene indicado por el valor de la variable `nprocs`. Esta zona paralela contiene todo el código que lleva a cabo el producto, por lo que en ningún caso será posible que existan más hilos que procesadores y por tanto haya una sobrexplotación de los recursos del sistema.

A continuación se invoca la función `omp_set_num_threads()`, esta función indica a la librería OpenMP el número de hilos debe crear, el cual en ningún caso sobrepasará el numero indicado en la directiva anterior. El número de hilos definidos por esta función se mantendrá hasta que se vuelva a invocar con un número distinto. La función se invoca indicando un número de hilos igual al número de filas de la matriz A sobre la que se invoca la función, o lo que es lo mismo, de la matriz que queda a la izquierda del operador "*".

Llegados a este punto se encuentra la directiva `#pragma omp for private(i)`. Esta directiva indica que se va a ejecutar un bucle `for` paralelo y en cual la variable `i` es privada para cada hilo, es decir, cada hilo ejecutará una iteración distinta del bucle. Este bucle será ejecutado, como máximo, por tantos hilos como iteraciones tenga, número que coincide con la cantidad de filas tiene la matriz A. En el caso de que este número sea mayor que el

número de procesadores de la máquina, el número de hilos será el de procesadores de la máquina.

Dentro del bucle se define una nueva zona paralela mediante la directiva `#pragma omp parallel`. Esta nueva zona hereda el número máximo de hilos por esta contenida dentro de la directiva inicial en que se definió este valor y, además, hereda el valor que se definió mediante la función `omp_set_num_threads`. Sin embargo este último valor se cambia en la siguiente línea invocando a la misma función con el valor del número de columnas de B . Como se puede observar el primer bucle indicaba las filas de A y este indica las columnas de B, por lo tanto se tiene que en caso de que haya suficiente procesadores se dedicara a cada fila de A y cada columna de B un hilo.

El último *pragma*, `#pragma omp for private(j,k,valor,total)`, indica que se va a ejecutar un bucle paralelo y que la variables `j`, `k`, `valor` y `total` son privadas para cada hilo, esta variables representan respectivamente, el índice del segundo bucle `for`, el índice del tercer bucle `for`, la variable que almacena el resultado de multiplicar un elemento de una fila de A por un elemento de una columna de B, y la variable que almacena la reducción de los valores anteriores, o lo que es lo mismo el valor final de cada elemento de la matriz resultado.

Finalmente cada vez que se completa una iteración del tercer bucle se almacena el valor calculado en la posición correspondiente de la matriz resultado. Cuando todos los valores han sido calculados la función termina y retorna el objeto `KFMatrix` que contiene el resultado.

Como se puede observar la paralelización con OpenMP de códigos ya implementados en forma secuencial es muy sencilla ya que apenas requiere cambiar el código secuencial, sino que con añadir directivas de compilador se puede obtener un código paralelo, sobre todo en el caso de bucles.

Una de las grandes ventajas de esta biblioteca es que una vez que se tiene el código paralelizado es posible compilarlo de forma paralela añadiendo en la línea de compilación el *flag* `-fopen` o en modo secuencial sin añadirlo.

Este código ha sido sometido a prueba para medir cuanto tiempo tarda en acometer la operación de producto de matrices. Para ello se han utilizado matrices de varios tamaños, más concretamente matrices cuadradas desde $2x2$ hasta $1024x1024$ en incrementos de potencias de 2. Estas se han comparado con los resultados de realizar las mismas operaciones de forma secuencial. Los resultados obtenidos se detallan en la

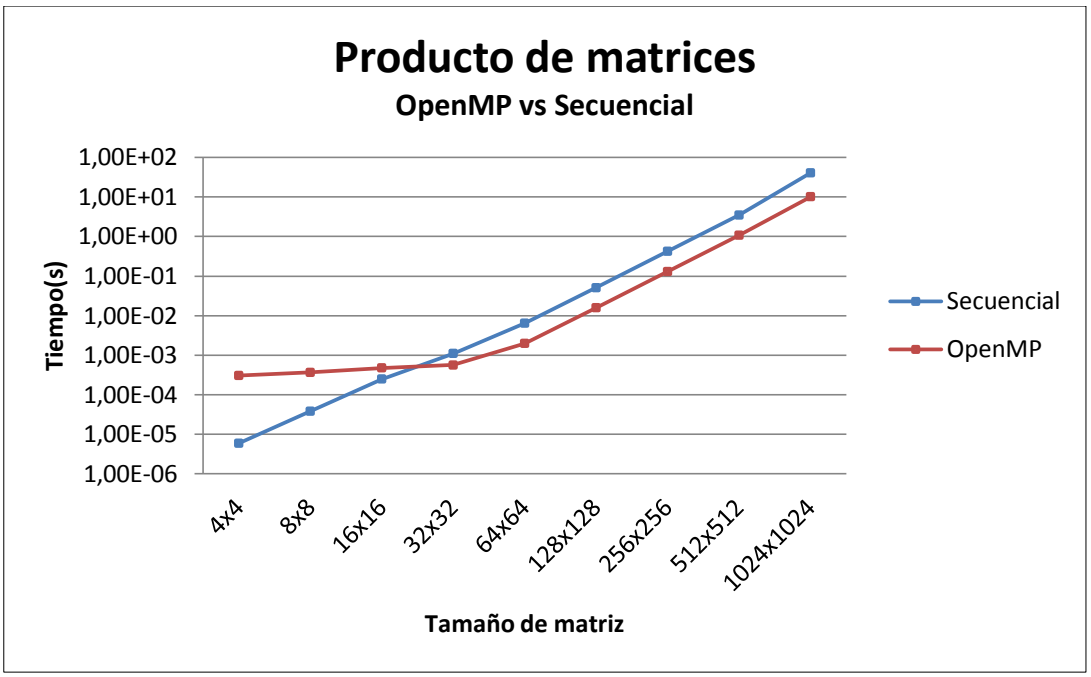

Figura 5.20. Tiempo de ejecución de producto de matrices OpenMP vs secuencial.

El gráfico muestra el tiempo empleado en multiplicar 2 matrices utilizando la función implementada en la Figura 5.19 ejecutándolo en modo secuencial y en modo paralelo con OpenMP. Como se puede observar el producto de matrices es más rápido en todos los casos cuyo tamaño es mayor o igual a $32x32$. Para tamaños de matriz grandes es muy recomendable su uso, ya que reduce el tiempo de cómputo significativamente.

Para tamaños de matriz pequeños es contraproducente el uso de esta técnica de paralelización ya que el resultado es peor que con una implementación secuencial. Esto es debido a que la creación, gestión y posterior destrucción de hilos conlleva un coste que en comparación con el tiempo que se gana en la realización del producto es muy elevado, aumentando el tiempo total de cómputo hasta sobrepasar el secuencial. Debido a que la operación de producto es la más costosa y que no conviene implementarla para un tamaño de $4x4$, el resto de operaciones menos costosas tampoco convendrá.

5.4 PARALELIZACIÓN CON ARBB

La otra solución que se ha llevado a cabo ha sido la paralelización de las operaciones algebraicas más costosas con Intel Array Building Blocks.

Como se ha explicado en el apartado 0 de este documento, ArBB es un framework en fase beta desarrollado por Intel y cuya característica principal es que permite paralelismo a nivel de datos además de a nivel de *thread*.

La pruebas que se ha realizado para comprobar si el uso de ArBB era beneficioso han sido sobre la función de producto de matrices. En la prueba que se ha llevado a cabo se multiplican dos matrices de varios tamaños.

El código utilizado para la primera prueba se muestra en la Figura 5.21.

```
void arbb_matrix_multiplication(
  /// input matrix A
  const arbb::dense<arbb::f64, 2>& a,
  /// input matrix B
  const arbb::dense<arbb::f64, 2>& b,
  /// output matrix C
  arbb::dense<arbb::f64, 2>& c)
{
  using namespace arbb;
  usize m = a.num_rows();
  usize n = b.num_cols();
  _for (usize i = 0, i < n, ++i) {
    dense<f64, 2> mult = a * repeat_row(b.col(i), m);
    dense<f64> col = add_reduce(mult);
    c = replace_col(c, i, col);
  } _end_for;
}
```

Figura 5.21 Codigo ArBB para multiplicar 2 matrices

En esta prueba se multiplican 2 matrices. La función recibe tres parámetros, los dos primeros de entrada y el tercero de salida, donde se almacenará el resultado. Estos parámetros son de tipo `arbb::dense`. Este tipo de ArBB, es un contenedor especialmente diseñado para aplicación de paralelismo a nivel de datos. Se implementa en forma de plantilla, la cual recibe dos parámetros. El primero es el tipo de dato que almacenará el contenedor, en este caso *`arbb::f64`*, el cual es un tipo de coma flotante con 64 bits de precisión. El segundo parámetro es el número de dimensiones de los datos, en este caso 2 por ser una matriz.

El algoritmo de multiplicación de matrices en ArBB es algo diferente de lo habitual debido a la forma de realizar las operaciones. ArBB aplica la misma operación sobre varios elementos del contenedor en una sola instrucción, el algoritmo de multiplicación de matrices, considerando la operación $A * B$, es el siguiente:

1. Multiplicar elemento a elemento A por una matriz temporal formada por la repetición de la primera columna traspuesta de B tantas veces como columnas tenga A.
2. Reducir las filas de la matriz temporal sumando sus elementos.
3. Sustituir la columna correspondiente de la matriz resultado por el resultado generado en el paso anterior.
4. Repetir los pasos 1, 2 y 3 hasta terminar con todas las columnas de B.

A continuación se muestra un ejemplo genérico del algoritmo multiplicando 2 matrices 2x2.

Se tienen dos matrices, A = $\begin{bmatrix} a_A & b_A \\ c_A & d_A \end{bmatrix}$ y B = $\begin{bmatrix} a_B & b_B \\ c_B & d_B \end{bmatrix}$ cuyo producto es el de la ecuación [5.2].

$$\begin{bmatrix} a_A & b_A \\ c_A & d_A \end{bmatrix} * \begin{bmatrix} a_B & b_B \\ c_B & d_B \end{bmatrix} = \begin{bmatrix} a_A * a_B + b_A * c_B & a_A * b_B + b_A * d_B \\ c_A * a_B + d_A * c_B & c_A * b_B + d_A * d_B \end{bmatrix} \quad [5.2]$$

Como el tamaño de las matrices es 2x2 se tendrán 2 iteraciones. Los pasos de la primera iteración son los siguientes:

Se calcula la matriz formada por la repetición de la primera columna de B traspuesta, [5.3].

$$\begin{bmatrix} a_B & c_B \\ a_B & c_B \end{bmatrix} \quad [5.3]$$

Se multiplica, elemento a la elemento la matriz A con la matriz [5.3].

$$\begin{bmatrix} a_A & b_A \\ c_A & d_A \end{bmatrix} * \begin{bmatrix} a_B & c_B \\ a_B & c_B \end{bmatrix} = \begin{bmatrix} a_A * a_B & b_A * c_B \\ c_A * a_B & d_A * c_B \end{bmatrix} \quad [5.4]$$

Se reduce el resultado de la ecuación [5.4] sumando los elementos fila a fila.

$$\begin{bmatrix} a_A * a_B + b_A * c_B \\ c_A * a_B + d_A * c_B \end{bmatrix} \quad [5.5]$$

Se sustituye la matriz [5.5], resultado del paso anterior en la primera columna de la matriz resultado.

$$\begin{bmatrix} a_A * a_B + b_A * c_B & 0 \\ c_A * a_B + d_A * c_B & 0 \end{bmatrix} \quad [5.6]$$

La segunda iteración contiene los mismos pasos que la primera pero esta vez se utiliza la segunda columna de B.

Se calcula la matriz formada por la repetición de la primera columna de B traspuesta, [5.7].

$$\begin{bmatrix} b_B & d_B \\ b_B & d_B \end{bmatrix} \quad [5.7]$$

Se multiplica, elemento a la elemento la matriz A con la matriz [5.8].

$$\begin{bmatrix} a_A & b_A \\ c_A & d_A \end{bmatrix} * \begin{bmatrix} b_B & d_B \\ b_B & d_B \end{bmatrix} = \begin{bmatrix} a_A * b_B & b_A * d_B \\ c_A * b_B & d_A * d_B \end{bmatrix} \quad [5.8]$$

Se reduce el resultado de la ecuación [5.4] sumando los elementos fila a fila.

$$\begin{bmatrix} a_A * b_B + b_A * d_B \\ c_A * b_B + d_A * d_B \end{bmatrix} \quad [5.9]$$

Se sustituye la matriz [5.9], resultado del paso anterior en la primera columna de la matriz resultado, obteniéndose el resultado final

$$\begin{bmatrix} a_A * a_B + b_A * c_B & a_A * b_B + b_A * d_B \\ c_A * a_B + d_A * c_B & c_A * b_B + d_A * d_B \end{bmatrix} \quad [5.10]$$

Como se puede observar se tiene que el resultado de la ecuación [5.2] es igual a [5.10].

Las pruebas que se han realizado siguiendo este algoritmo se han hecho sobre matrices cuyos tamaños varían desde 2x2 hasta 1024x1024 en potencias de dos. Se han realizado suficientes ejecuciones en cada caso como para conseguir una medida de tiempo fiable.

El resumen de los tiempos tomados para la multiplicación de matrices, tanto en su forma secuencial como con ArBB se muestra en la Figura 5.22.

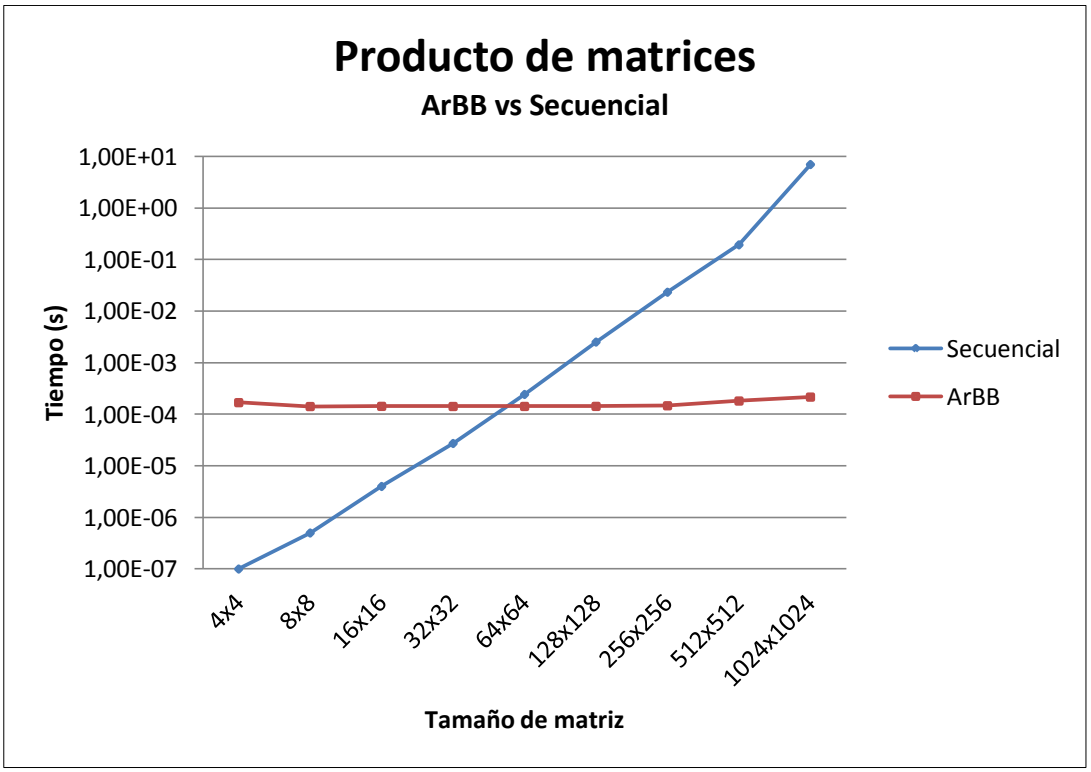

Figura 5.22. Tiempo de ejecución de producto de matrices secuencial vs ArBB

Como se puede observar en el gráfico ArBB no es eficiente para tamaños de matriz menores de 64x64. A partir de estos tamaños es muy recomendable su uso ya que reduce el tiempo de cómputo en gran medida.

La razón de que para tamaños más pequeños sea más lento que las versiones secuenciales se encuentra en la sobrecarga que genera tanto el compilador JIT como la creación de hilos. Esto produce que no sea viable implementarlo en el proyecto ya que ralentizaría las operaciones algebraicas, tanto el producto, como ya se ha demostrado, como el resto de operaciones que son menos pesadas y por tanto menos dadas a su paralelización.

5.5 DESARROLLO DEL SISTEMA DE COLAS

Como paso previo a la consecución de un sistema capaz de tratar datos de varios objetos móviles, ha sido necesario diseñar e implementar un sistema de gestión de datos que organizase los mismos de forma que se pudiesen obtener ordenados y de la manera más rápida y eficiente posible, así como que permitiese insertar los nuevos datos que llegan desde los sensores o el simulador de forma concurrente.

Este sistema de gestión se ha implementado como un conjunto de colas concurrentes. Cada una de estas colas está asociada a un objeto móvil a través de un identificador numérico único, y almacena todos los datos de medidas correspondientes a ese objeto móvil.

Para conseguir acceso a las colas de datos se ha implementado una cola circular de punteros. Cada uno de estos punteros apunta a una de las colas de almacenamiento de datos.

5.5.1 COLA CIRCULAR DE PUNTEROS

El acceso a las colas de datos debe hacerse de forma que todas sean tratadas por igual, ya que es necesario que se traten los datos de todos los objetos móviles que se estén monitorizando. La solución que se ha implementado es una cola que contiene punteros a las colas de datos. Esta cola se recorre de forma circular, tratando el primer dato contenido en la cola de datos referenciada por el puntero. De esta forma se tratan datos de todas las colas sin dar prioridad a unas sobre otras.

Esta cola esta implementada sobre un contenedor de tipo `tbb::concurrent_queue`. Este contenedor está incluido en la biblioteca TBB y su principal característica es que es un contenedor concurrente. La razón de utilizar un contenedor concurrente es la necesidad de extraer e insertar los punteros en él constantemente como se verá en el punto 5.5.4 de este documento. El tipo `tbb::concurrent_queue` es un contenedor lineal, no circular, sin embargo se presenta como circular ya que se recorre de esta forma. Cada vez que un hilo obtiene acceso a una cola de datos se avanza una posición en la cola de punteros, si se detecta que la posición es la última se vuelve a la primera posición, consiguiendo así el efecto circular.

5.5.2 COLAS DE DATOS

Las colas de datos son las encargadas de almacenar los datos de medidas que generan los sensores para su posterior tratamiento. Esta colas, al igual que la cola circular, están implementadas utilizando el contenedor de TBB `tbb::concurrent_queue`, ya que también se necesita insertar y extraer datos de ellas de forma concurrente. Estas colas están instanciadas en la clase `KFTaskManager` y contienen objetos del tipo `KFData`, detallado en el punto 4.4.2.1. La Figura 5.23 muestra el sistema de colas.

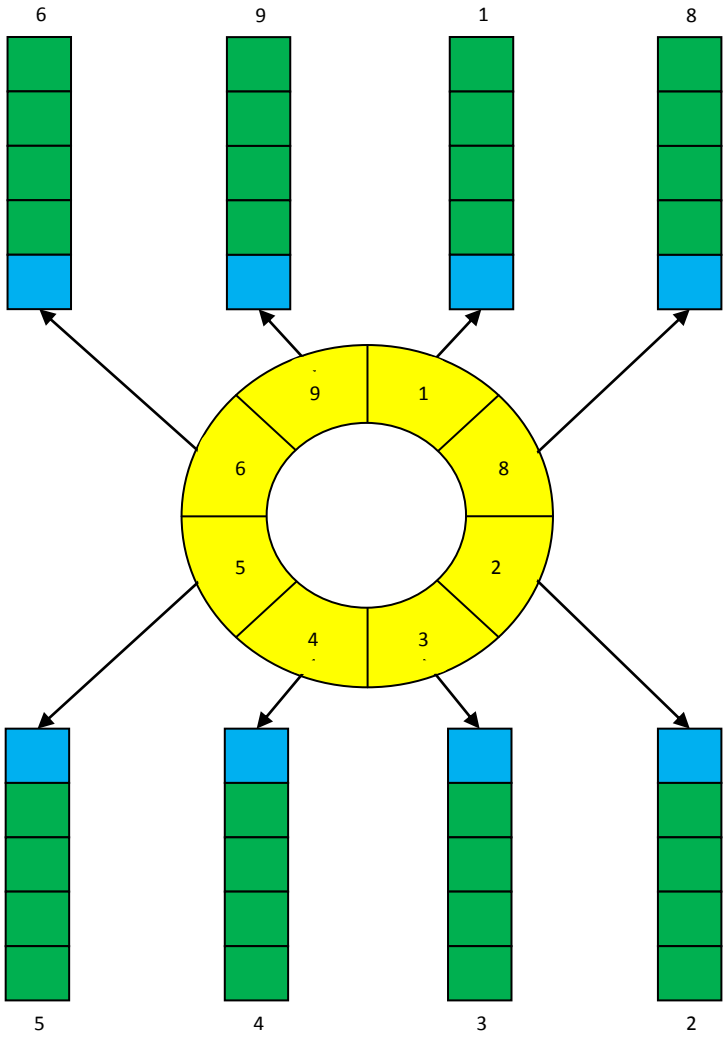

Figura 5.23 Representación del sistema de colas

5.5.3 INSERCIÓN DE DATOS

Durante la ejecución del sistema se reciben nuevos datos por parte de los sensores. En el caso de este proyecto estos datos se insertan desde un fichero en el instante aproximado que indica su marca de tiempo, el detalle de esta operación se puede ver en el punto 4.4.3 de este documento.

La inserción de datos se realiza a través de la función `addTask` de la clase `KFTaskManager`. Esta función recibe como parámetros una referencia a un vector con los datos a insertar y un identificador del objeto móvil al que pertenecen. Primeramente se comprueba si ya existe una cola de datos para ese objeto, si existe se insertan los datos y se termina, si no es así se crean las estructuras necesarias antes de insertar los datos. El diagrama de este algoritmo puede verse en la Figura 5.24. Las estructuras que deben crearse son una cola de datos para el objeto móvil, `KFQueue`, un objeto `KFLib` donde se guardará el estado del objeto y un puntero a la cola de datos que se insertará en la última posición de la cola circular. Las dos primeras estructuras se insertarán en contenedores ordenados con el fin de poder encontrarlas rápida y fácilmente.

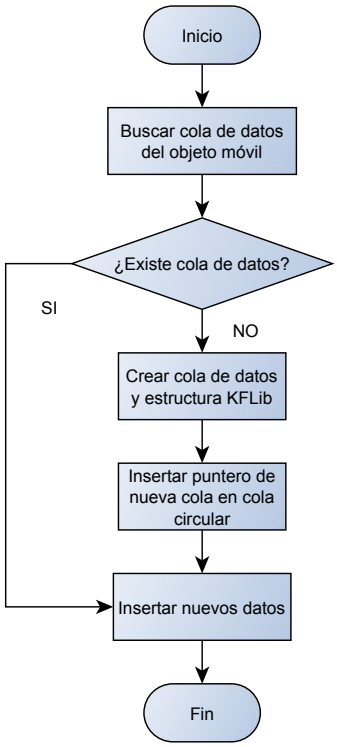

Figura 5.24 Diagrama del algoritmo de inserción.

Los contenedores donde se almacenan los objetos de tipo `KFQueue` y `KFLib` son contenedores asociativos de la biblioteca estándar de plantillas de C++, más concretamente del tipo `std::map`. Estos contenedores almacenan los objetos mencionados junto con una clave de indexación, que en este caso es el identificador del objeto móvil. La ordenación se hace en un árbol rojinegro por lo que la búsqueda de una clave y su consiguiente valor u objeto asociado se hace en tiempo logarítmico.

El acceso a las colas de datos a través de este contenedor se realiza solamente cuando se insertan datos, ya que para extraer datos se hace a través de los punteros almacenados en la cola circular antes descrita como se explicará en el punto 5.5.4.

El acceso a los objetos `KFLib` se realiza cada vez que un hilo trata un dato, ya el resultado parcial de cada objeto móvil se almacena en atributos de esta clase, además, de que esta clase representa el propio algoritmo de los filtros de Kalman y, por tanto, implementa las funciones que permiten su aplicación.

Los objetos de tipo `std::map` no son concurrentes a diferencia de los anteriormente presentados. El hecho de que se haya elegido este tipo de contenedor y no otro concurrente de TBB es que no existe ningún contenedor asociativo y concurrente al mismo tiempo y, que por tanto, además de permitir el acceso de varios hilos simultáneamente, permitiese búsquedas rápidas sobre los datos almacenados. Para solucionar el problema de acceder de forma concurrente a estos contenedores se han utilizado *mutex*. Cada vez que se va a buscar o insertar un objeto en uno de estos contenedores se debe coger un *mutex,* evitando así las condiciones de carrera entre diferentes hilos. Obsérvese que el bloqueo que produce el *mutex* es sobre el contenedor, no sobre los datos, es decir, en al caso de la consulta se producirá un bloqueo el tiempo necesario para buscar y obtener el dato almacenado, no se bloqueará el acceso mientras se esté trabajando con el dato.

La inserción de datos debe hacerse teniendo en cuenta las marcas de tiempo de los datos cargados de fichero. En una situación con sensores reales colocados en diferentes puntos, las medidas tomadas no llegarían al sistema en el momento justo de su toma sino que, debido a diversos factores, como retardos en el tratamiento de los datos por parte de los sensores, retardos en el canal de comunicación, etc, llegarían con cierto retraso. Para simular esto la ejecución de la función `putIterationData` que introduce los datos leídos de fichero se realiza periódicamente, insertando cada vez los datos correspondientes a ese período de

tiempo como se detalla en el apartado 4.4.3. Esto se realiza con un hilo aparte como se verá más adelante en el documento.

5.5.4 OBTENCIÓN DE DATOS

La obtención de datos o tareas se realiza a través de la función `getTask` de la clase `KFTaskManager`. El funcionamiento de esta función gira en torno a la cola circular de punteros. Debido a la naturaleza iterativa del algoritmo se debe garantizar que en ningún caso se trata un dato de un objeto móvil antes de que se hayan terminado de tratar los que van antes que él en su cola. Esta restricción obliga a idear un mecanismo que permita identificar y/o bloquear, de alguna forma, aquellas colas cuyos datos deben esperar a que un dato anterior en orden a ellos y del mismo objeto móvil, y por tanto de la misma cola, termine de ser tratado por completo.

Para solventar este problema se ha optado por extraer de la cola circular los elementos que apuntan a una cola que deba esperar a que un dato se trate, de esta forma ningún hilo más podrá obtener una referencia a esta cola y por tanto no podrá comenzar el tratamiento de un dato de la misma. Una vez que el dato en cuestión ha sido tratado por completo se vuelve a insertar el puntero al final de la cola circular. En el caso en que la cola cuyo puntero se ha obtenido no tenga ningún dato disponible se vuelve a insertar el puntero en la cola circular. Este proceso se repite indefinidamente como se muestra en la Figura 5.25

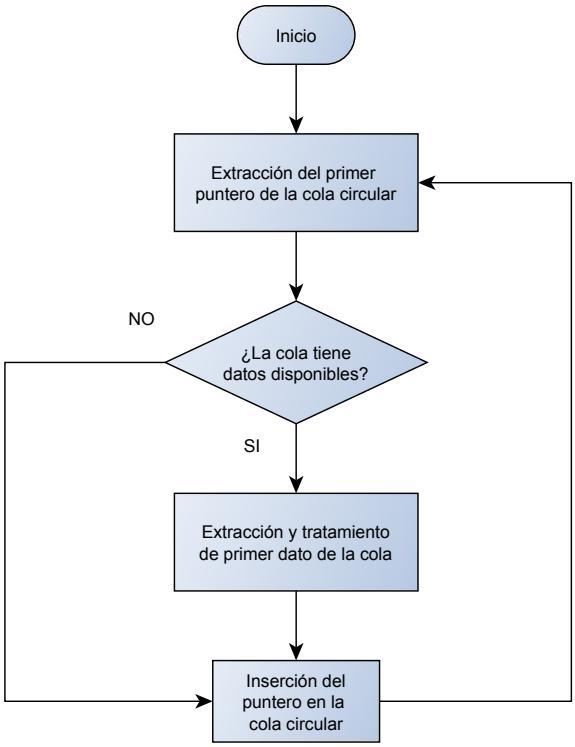

Figura 5.25 Algoritmo de extracción e inserción de tareas

Esta forma de obtener las tareas o datos a tratar permite reducir el tiempo de bloqueo al mínimo, ya que sólo es necesario bloquear la cola circular el tiempo justo para extraer un puntero y el tiempo justo para volverlo a insertar. El resto del tiempo, mientras se trabaja con el dato extraído, la cola circular esta libre para continuar haciendo su trabajo con el resto de hilos del sistema en caso de que haya más de uno.

El bloqueo necesario para extraer y posteriormente insertar no necesita de elementos de sincronización explícitos, como mutex, semáforos, etc, ya que para implementar la cola circular de punteros se ha utilizado un contenedor concurrente incluido en TBB que se encarga de bloquear cuando es necesario el acceso para evitar condiciones de carrera. Este contenedor concurrente no tiene forma circular, sino lineal. Para producir el efecto de cola circular sobre un contenedor lineal, se tiene un iterador que va avanzando desde el principio hasta el final del contenedor, volviendo otra vez al principio cuando llega al final.

5.6 ALGORITMOS DE DESCARTE DE DATOS

Debido a que a priori no se conoce la cantidad de datos que pueden llegar en un momento determinado es necesario decidir qué criterios se van a utilizar para descartar los datos que el sistema no pueda tratar. El motivo principal para descartar datos que el sistema no pueda tratar es que este no se retrase demasiado, es decir, que en un instante de tiempo no se estén tratando datos que pertenece a un instante demasiado antiguo, ya que esto produciría que los resultados que se van obteniendo fuesen obsoletos.

5.6.1 ALGORITMO I

El primero de los algoritmos se basa en vaciar o limpiar las colas de datos cada cierto tiempo. De esta forma se asegura que el sistema nunca tratará datos cuya antigüedad sea mayor que el tiempo que pasa entre vaciado y vaciado. Para la ejecución de este método de descarte se necesita la ejecución de un hilo dedicado, ya que se debe se debe asegurar que el limpiado se realiza cada cierto tiempo, este tiempo o ciclo de ejecución es configurable.

La función que implementa el algoritmo recibe como parámetro el objeto de tipo `KFTaskManager`, el cual contiene las instancias de las colas de datos del sistema. La primera vez que se ejecuta el algoritmo toma el tiempo del sistema, este tiempo se utilizará como referencia durante el resto de la ejecución. Posteriormente, el algoritmo entra en un bucle infinito donde ejecuta su funcionalidad. En este bucle se ejecuta la función `clearQueues` del objeto de tipo `KFTaskManager` que recibe por parámetro, esta función es la encargada de vaciar todas las colas de datos del sistema recorriéndolas una a una y eliminando los datos que contienen.

Posteriormente se suma al tiempo que se ha tomado al inicio, el tiempo que se ha definido como ciclo de ejecución, esta suma nos da el instante en el cual se deberá volver a ejecutar la limpieza de colas, por lo que dormiremos el hilo hasta que llegue ese momento. Debe tenerse en cuenta que dado que el sistema es un sistema de calidad de servicio puede darse la situación de que la ejecución de la función tome más tiempo que el definido en el ciclo de ejecución, esto provocará un retraso en el tratamiento de datos y por tanto una pérdida de calidad en el resultado, pero en ningún caso produciría un fallo de ejecución en el sistema.

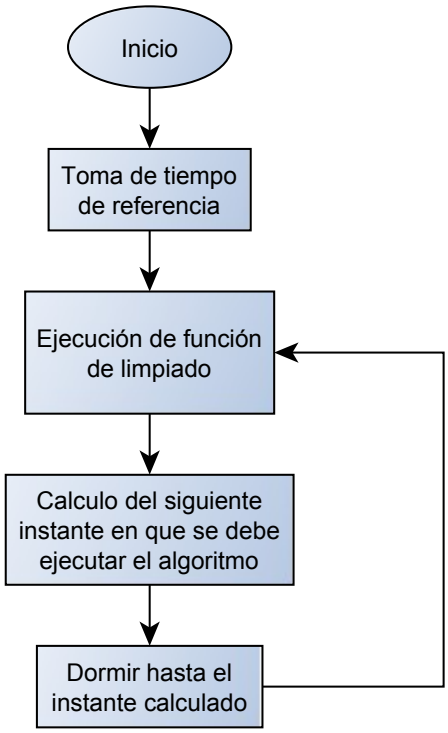

Figura 5.26 diagrama del algoritmo de vaciado de colas de datos

Cabe destacar que este algoritmo de descarte es necesario ejecutarlo en un hilo propio que despierte cada cierto tiempo y realice el descarte. Este hilo pasa la mayor parte del tiempo durmiendo y apenas supone sobrecarga para el sistema, aunque dado que debe acceder a recursos compartidos, concretamente a las colas de datos, debe sincronizarse con el resto de hilos. Este inconveniente se resuelve en parte porque las colas de datos están implementadas con contenedores concurrentes como ya se ha explicado en el punto 5.5.2 del documento. Sin embargo el contenedor donde se guardan estas colas de datos, `_queues` de la clase `KFTaskManager`, no es concurrente y dado que se debe acceder para obtener las colas es necesario sincronizar este acceso, ya que este contenedor puede estar siendo accedido por otros hilos. Para solventar este problema se utilizan los mutex de la clase `KFTaskManager`, en concreto el mutex `_m2` que controla el acceso al contenedor donde están almacenadas las colas.

5.6.2 ALGORITMO II

El segundo algoritmo que se ha implementado se basa en realizar un descarte ajustándose al retraso del sistema en cada momento. La idea subyacente es ajustar el número de elementos descartados a las necesidades reales de descarte en cada momento. Por otro lado también se busca una forma de evitar los saltos que se producían con el algoritmo anterior, ya que producen que la trayectoria pierda linealidad y por tanto que el filtro no converja adecuadamente.

Este algoritmo no necesita un hilo propio para poder funcionar sino que se ejecuta en la propia rutina de tratamiento de datos, evitando tener que sincronizar otro hilo que accede con operaciones de escritura a datos compartidos, evitando por tanto bloqueos del programa y esperas.

Para ajustarse a la situación del sistema es necesario adoptar un criterio para medirlo. En este caso el criterio que se ha adoptado ha sido el retraso que tiene el sistema en el tratamiento de los datos recibidos, es decir, para un instante de la ejecución se deberá comprobar que marca de tiempo tiene el dato que se va a tratar y comprobar la diferencia entre el instante actual y la marca de tiempo del dato. Dependiendo del valor de esta operación se realizará un mayor o menor descarte.

A diferencia del algoritmo anterior, en el cual el descarte se realizaba en bloques de datos, en este algoritmo se realiza de forma distribuida a lo largo de los datos. Cuando se detecta, al tratar un dato, que existe un retraso en el tratamiento, se descartan dependiendo del valor de este retraso, un número determinado de elementos de la cola a la que pertenezca el dato que se va a tratar. Esto se repite por cada dato que se trata, de esta forma cada vez que se trata un dato se obtiene información sobre el estado del sistema, permitiendo alterar la cantidad de descartes en tiempo real.

Los valores de descarte en función del retraso se han seleccionado a tenor de resultados experimentales y son los contenidos en la siguiente tabla.

Retraso del sistema (ms)	Datos descartados
< 250	0
>= 205 < 500	25
>= 500 < 750	40
> 750	50

Tabla 5.1 Valores de descarte según retraso del sistema

El algoritmo implementado se representa en la Figura 5.27.

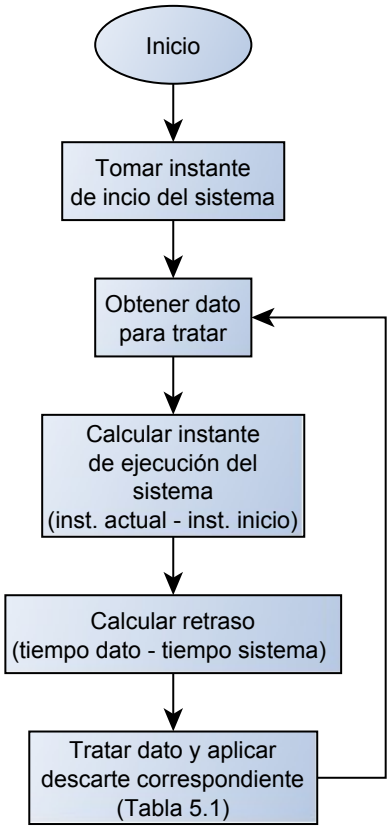

Figura 5.27. Representación del algoritmo de descarte

Al comienzo de la primera ejecución se toma el tiempo, este tiempo inicial se tomará como referencia para calcular el instante en que se encuentra el sistema cada vez que se trate un dato. Posteriormente se entra en un bucle que se ejecutará constantemente. Dentro de este bucle se llevan a cabo todas las operaciones, tanto de tratamiento de datos como de descarte de los mismos. Al comienzo de cada iteración del bucle se obtiene el dato que se va a tratar en esa iteración, posteriormente se toma el tiempo del sistema y se calcula el instante

de ejecución en que se encuentra, este valor se compara con la marca de tiempo que tiene el dato que se va a tratar y se calcula el retraso del sistema. Dependiendo del valor del retraso se descartan, de la cola a la que pertenece el valor que se va a tratar, los elementos correspondientes (indicados en la Tabla 5.1). Por último se trata el valor obtenido al principio y se vuelve a ejecutar el bucle.

Este algoritmo se ejecuta directamente en la función que trata los datos por lo que no es necesario tener un hilo dedicado para él.

5.6.3 ALGORITMO III

El tercer algoritmo de descarte se basa en la capacidad de tratamiento del sistema. La idea de este algoritmo es adecuarse a la capacidad de cómputo de datos del sistema donde se ejecuta. Para adaptarse a la situación del sistema se tiene en cuenta un histórico de este en cuanto al tratamiento de datos. El número de datos descartados se realiza en base al número de elementos contenidos en la cola sobre la que se va a realizar el descarte, esto evita que el descarte afecte más a los objetos cuya cantidad de medidas es menor y por lo tanto evita la pérdida de precisión.

La unidad que se utiliza para medir la capacidad del sistema es el número de elementos tratados anteriormente. Al saber que número de elementos se pueden tratar y qué cantidad espera para ser tratada se puede hacer una aproximación del estado del sistema y actuar convenientemente.

Al contrario que en el caso anterior, este algoritmo necesita un hilo dedicado, ya que es necesario poder tomar datos, cada cierto tiempo, del número de elementos que se han tratado y descartado para poder realizar el histórico que permite saber cuántos datos es capaz de tratar el sistema. El número de datos que es capaz de tratar el sistema se calcula como la media de datos por objeto móvil que el sistema ha sido capaz de tratar en los últimos 10 segundos. De esta forma se puede conocer, para cada cola de datos, cuantos datos se podrán tratar y cuantos se deberán descartar. Para poder calcular la media de datos tratados por el sistema es necesario llevar la cuenta de los mismos, este contador se implementa con un tipo atómico, ya que deberá ser accedido por dos hilos, uno de ellos que lo incrementará cuando se trate un dato y otro que leerá su valor y lo reiniciará.

El descarte se realiza de forma distribuida a lo largo de los datos de cada cola, evitando las discontinuidades que hacen que el algoritmo no converja rápido y bien. El valor de descarte se calcula en función del histórico de datos tratados por el sistema y del tamaño de la cola sobre la que se va a realizar el descarte. Para decidir qué número de elementos se va a descartar se calcula un factor de descarte, o número de elementos descartados por cada elemento tratado. El cálculo de este factor de descarte se hace cada vez que se recalcula el histórico, su valor es el resultado de dividir el número de elementos de cada cola entre el valor medio de elementos que se pueden tratar por cada cola.

Cada cola de datos tiene su propio factor de descarte y estos deben estar almacenados de forma que tanto el hilo que trata los datos como el que calcula la media histórica puedan acceder a ellos concurrentemente. La solución por la que se ha optado para almacenar y poder acceder de forma concurrente a los factores de descarte ha sido un vector de enteros atómicos. La forma de almacenar los datos en el vector es a través de su id, el cual se utiliza para indexar la posición del vector donde está almacenado el valor de factor de descarte que le corresponde. Dado que el valor almacenado es un tipo atómico es posible acceder de forma concurrente al mismo evitando condiciones de carrera.

El factor de descarte de cada cola es calculado por el hilo que trata los datos. Su cálculo debe hacerse cada vez que se recalcula el valor de la media de datos tratados. Dado que el hilo que calcula la media de datos y el que calcula el factor de descarte son hilos diferentes es necesario señalizar de alguna forma el recalculo de la media por parte del primero para que el segundo recalcule los factores de descarte. Esta señalización se realiza en los propios valores del factor de descarte. Cuando se recalcula la media de datos tratados cambia el factor de descarte de todas las colas de datos al valor especial -1, este valor no se puede dar como factor de descarte. Cuando el hilo encargado de tratar los datos y de calcular el factor de descarte obtenga un dato para ser tratado y compruebe que su factor de descarte es -1 recalculará el mismo, manteniéndolo así actualizado.

Como ya se ha dicho, para implementar el algoritmo se necesitan un hilo independiente que recoge los datos de tratamiento y calcula la media histórica además del hilo que trata los datos, el cual aplica el descarte y recalcula el factor de descarte de cada objeto cuando es necesario. En la siguiente figura se representan los diagramas de ambas funciones.

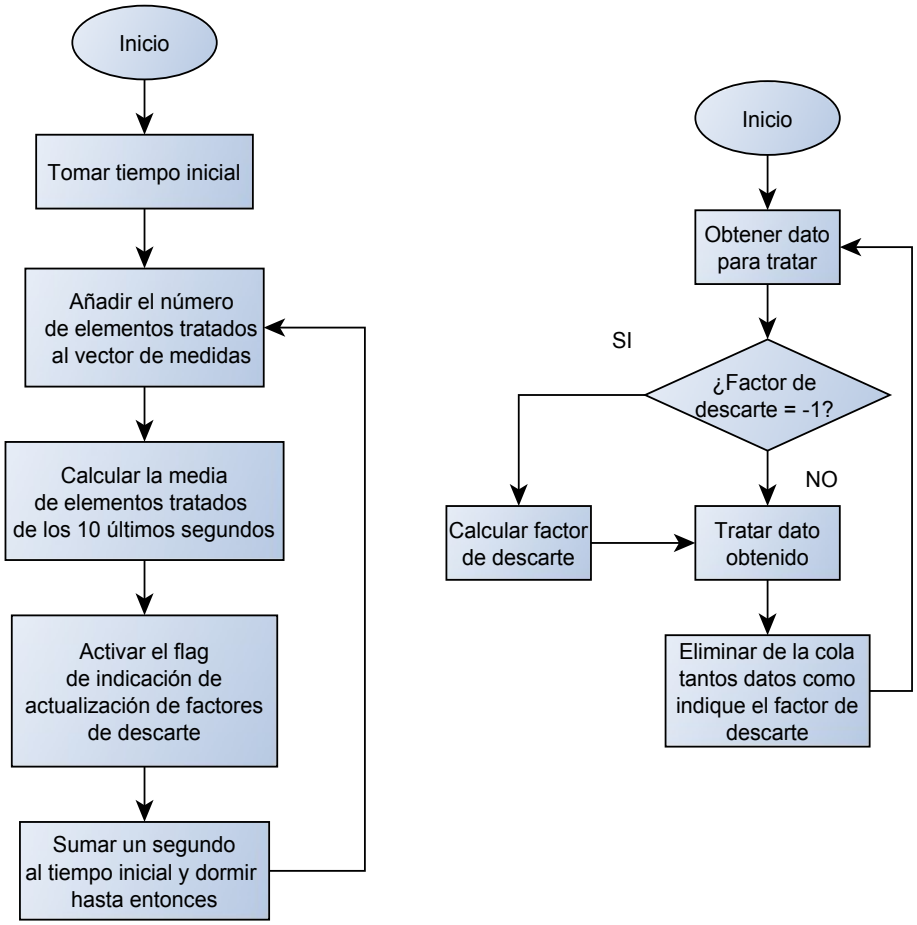

Figura 5.28. Diagramas de las funciones que implementan el tercer algoritmo de descarte

La Figura 5.28 representa las dos funciones que implementan el algoritmo de descarte. El diagrama de la izquierda representa la función que se encarga de calcular la media de datos tratados. El de la derecha representa la función de tratamiento de datos que además implementa el descarte de los mismos y el cálculo de los factores de descarte.

La función que calcula la media de datos tratados, lo hace en base a los diez últimos segundos de ejecución del sistema. Para poder saber cuántos datos se han tratado, cada vez que se trata un dato se incrementa un contador atómico global. Cada 1 segundo se ejecuta la función que calcula la media, toma el valor del contador anterior y lo reinicia a cero. Este valor se añade a un vector y posteriormente se calcula la media de datos tratados con los 10 últimos valores almacenados en el vector, de esta forma obtenemos el valor de datos tratados por el sistema en los 10 últimos segundos. Este valor se divide entre el número de colas de datos, o lo que es lo mismo entre el número de objetos móviles monitorizados por el sistema,

obteniéndose así el número medio de datos tratados por cola u objeto móvil. Posteriormente se cambia el factor de descarte de todas las colas al valor -1, este valor indica que la media histórica ha sido recalculada y por tanto el factor de descarte anterior es inválido y hay que volver a calcularlo. Finalmente el hilo se duerme hasta que llega el momento de volver a ejecutar la función.

La función que trata los datos y realiza el descarte, representada en la parte derecha de la Figura 5.28, trata los datos y descarta parte de ellos en función de los valores calculados anteriormente. Primeramente obtiene una cola de datos para tratar. Comprueba el valor del factor de descarte de la cola de datos obtenida, si es -1 quiere decir que es necesario recalcular este valor dividiendo el número de datos almacenados de la cola obtenida entre la media de datos tratados por cola. El valor obtenido en la operación anterior se redondea al entero mayor más cercano, es decir, se le aplica la que habitualmente se conoce como función techo. Esto se hace por dos motivos:

1. El valor debe ser entero ya que representa el número de elementos que se descartarán por cada elemento tratado.
2. No puede ser el entero inmediatamente inferior, ya que al descartar menos valores que el factor de descarte se acumularían valores que no da tiempo a tratar. En caso de que el valor de dividir en número de elemento en la cola entre la media sea menor que 1, el valor del factor de descarte será 0, ya que estaremos en la situación en que es posible tratar más elementos de los que hay almacenados en la cola y, por lo tanto, no es necesario descartar ninguno.

Tras calcular el factor de descarte, en caso de que sea necesario, se pasa a tratar el primer dato de la cola y posteriormente a eliminar tantos elementos como indique el factor de descarte.

Como se puede observar, este algoritmo tiene en cuenta tanto el estado, o capacidad, del sistema en el momento de tratar los datos, como las diferencias en el número de datos de cada objeto móvil. Esto permite realizar un avance importante, sobre todo en situaciones en que se tengan objetos móviles con un número elevado de medidas y otro con un número reducido, y además el sistema se sature y empiece a descartar datos, ya que este algoritmo descartará en mayor medida las medidas pertenecientes a colas con mucho elementos, sin

embargo, descartará pocos datos de colas con pocos elementos, evitando así la perdida de precisión en estos últimos debido a que finalmente se tengan pocos datos sobre ellos.

5.7 FUNCIÓN AUXILIAR DE INSERCIÓN DE DATOS

Para la ejecución del sistema ha sido necesario implementar una función que se encargue de decidir cuándo se insertan los datos. La función que se ha implementado se llama `insertar` y permite simular la llegada de datos en tiempo real. Esta función recibe como parámetro una referencia a un objeto de la clase `DataLoader` y un valor entero positivo y no retorna ningún valor.

La función se ejecuta en un hilo independiente. Su única actividad es invocar la función `putIterationData` de la clase `DataLoader` y dormir el hilo hasta que llegue el momento de volverla a invocar, este tiempo entre invocación e invocación es configurable por el usuario.

Cabe destacar que la ejecución de esta función es prioritaria ya que si tuviese que esperar demasiado a que hubiese recursos libres el sistema podría quedarse sin datos que tratar y cuando estos llegasen ya sería tarde, quedando obsoletos o siendo descartados, por lo que aunque se permite que en ocasiones se retrase, ya que como se ha dicho nos encontramos en un sistema de tiempo real acrítico y por tanto flexible en tiempos, se debe intentar que esto ocurra lo menos posible.

El funcionamiento de la función se detalla en la siguiente figura.

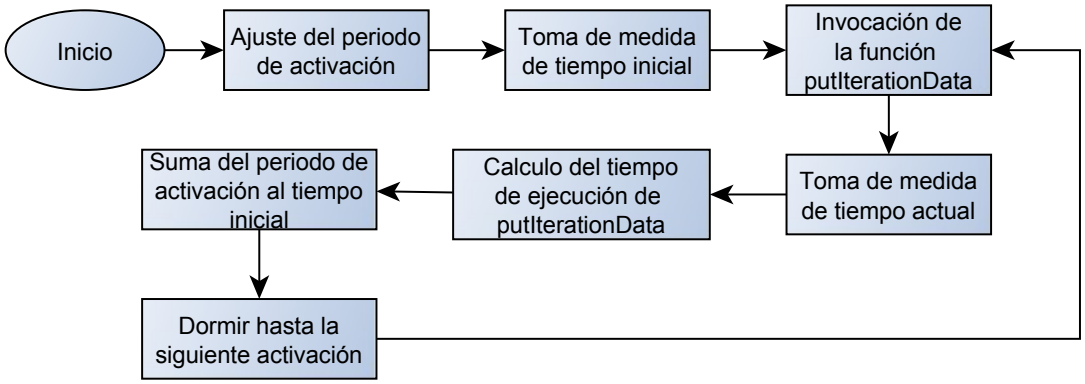

Figura 5.29. Diagrama de la función insertar.

Al comienzo de la función se ajusta el valor del periodo de activación, este valor es un entero sin signo que se recibe por parámetro y que indica cada cuantos milisegundos se debe ejecutar la función. A continuación se toma el tiempo actual del sistema. Este valor servirá

como referencia durante toda la ejecución, ya que a partir de él se calcularán los instantes en que se debe volver a ejecutar la función.

Seguidamente la función entra en un bucle que se ejecutará de forma infinita. Este bucle realiza las siguientes operaciones:

1. Ejecuta la función `putIterationData` de la clase `DataLoader`, como ya se ha indicado y explicado en el punto 4.4.3 esta función inserta en las colas los datos cuya marca de tiempo está comprendida entre el instante actual y el instante de la última vez que se ejecuto la función.
2. Terminada la ejecución de la función `putIterationData` se vuelve a tomar el tiempo del sistema y se calcula el tiempo transcurrido en ejecutarla restándole el tiempo inicial. Lo ideal es que este tiempo sea menor que el periodo de activación. Si es mayor se notificará el suceso con un mensaje por pantalla.
3. Se calcula el instante de tiempo en que la función se deberá volver a ejecutar, esto se consigue sumando al tiempo inicial el periodo de activación.
4. Una vez que se ha calculado el instante de la próxima activación se duerme el hilo hasta que llegue este instante, momento en que despertará y ejecutará el paso 1 del bucle.

5.8 AJUSTE DE PARÁMETROS DE PLANIFICACIÓN DE LOS HILOS

Como se ha podido ver en los últimos puntos, existen tres tareas que se deben ejecutar en el sistema, inserción de datos en las colas, tratamiento de los mismos y descarte según el algoritmo correspondiente. Cada una de estas tareas se implementa en una función independiente, exceptuando el descarte del algoritmo II que se implementa en la misma función que el tratamiento. Cada una de estas funciones es ejecutada por uno o varios hilos.

El hecho de que haya varios hilos y que estos compartan datos produce que se necesite sincronización y por lo tanto sucedan bloqueos. Los bloqueos en el acceso a datos son inevitables y suceden sobre los distintos hilos de forma más o menos aleatoria, sin embargo hay hilos que no deberían esperar, o al menos esperar el menor tiempo posible, para acceder a una variable compartida. Para estos su correcto funcionamiento está condicionado al cumplimiento de unos plazos temporales que, aunque son flexibles, se debe intentar cumplir en la medida de lo posible.

Para poder minimizar el tiempo de espera de los hilos que deban cumplir plazos temporales es necesario definir cuáles son estos hilos y cuál es su orden de prioridad en la ejecución del sistema. Entre las tres funciones que implementan las tareas del sistema existen dos que deben cumplir plazos, estas son la función de inserción de datos y la función de descarte en los casos en que existe. Además estas funciones no deben ser interrumpidas durante su ejecución ya que esto propiciaría que no se inserten o descarten todos los datos que debiesen.

Debido a que hay tres funciones se han especificado y asignado a las tareas tres niveles de prioridad como se indica en la Tabla 5.2.

Tarea	Nivel de prioridad
Inserción de datos	3
Descarte de datos (Algoritmos I y III)	2
Tratamiento de datos	1

Tabla 5.2. Prioridades asignadas a las tareas

La tarea de inserción de datos, cuya prioridad es la más alta, es la tarea cuya criticidad es mayor, ya que la no ejecución o la ejecución de forma muy retrasada de la misma puede suponer que el sistema trate datos con mucho retraso o incluso su pérdida dependiendo del algoritmo de descarte, por tanto es importante que esta tarea siempre tenga prioridad sobre el resto.

La segunda tarea en nivel de prioridad es la encargada de realizar el descarte de datos (en los casos en que exista). Esta tarea también debe cumplir plazos temporales aunque su retraso es menos crítico que en el caso de la inserción. Sin embargo esta tarea debe ser más prioritaria que la tarea de tratamiento de datos ya que si no se ejecuta a tiempo la tarea de tratamiento puede llegar a tratar datos obsoletos y por tanto se estarán desperdiciando recursos del sistema. La tarea de descarte debe realizar su ejecución sin ser interrumpida ya que de otra manera se dejarían de descartar datos de algunas colas, más concretamente y por la implementación del sistema este fenómeno se daría sobre las últimas colas creadas.

La última tarea en cuanto a prioridad es la tarea encargada de tratar los datos. Esta tarea debe ejecutar siempre que existan recursos libres para ello, pero dado que no debe cumplir ningún plazo temporal nunca debe entorpecer la ejecución de tareas que si deban cumplirlo y por tanto en caso de que no haya recursos disponibles y una tarea de mayor prioridad se active, la tarea de tratamiento deberá ceder su uso del sistema y por tanto podrá ser interrumpida.

Como se ha visto las tareas de inserción y descarte de datos deben ejecutar de forma que no puedan ser interrumpidas y por el contrario la tarea de tratamiento podrá ser interrumpida si es necesario. Para solventar esta situación a las dos primeras tareas se les asignará una planificación de tipo FIFO y a la tercera una de tipo *Round-Robin*. El planificador de Linux soporta ambos tipos de planificación por lo que no será necesario realizar ningún cambio o implementación extraordinaria.

El procedimiento que se ha seguido para ajustar los parámetros de prioridad y planificación del sistema operativo sobre cada uno de los hilos está basado en la interfaz que ofrece POSIX. La Figura 5.30 muestra el código utilizado para llevar a cabo el ajuste de estos parámetros. Este código se ejecuta al principio de cada una de las 3 funciones descritas, ejecutando el resto del código con la planificación ajustada.

```
sched_param sch;
int policy;
pthread_getschedparam(pthread_self(), &policy, &sch);
sch.sched_priority = 3;
policy = SCHED_FIFO;
pthread_setschedparam(pthread_self(), policy, &sch);
```

Figura 5.30. Código de ajuste de parámetros de planificación y prioridad.

Como ya se ha dicho la planificación contempla dos parámetros, la política de planificación y la prioridad del hilo. En el código anterior se modifican estos dos parámetros de la siguiente forma:

1. Se declaran dos variables. La primera es una estructura de tipo `sched_param`, esta estructura está formada por un atributo de tipo entero que define la prioridad del proceso. Su nombre es `sched_priority`. La segunda variable es un entero que define la política de planificación del hilo. Esta variable puede tomar 5 valores, `SCHED_FIFO`, `SCHED_RR` para planificación de tiempo real y `SCHED_OTHER`, `SCHED_BATCH` y `SCHED_IDLE` para planificación normal, todos ellos constantes definidas en POSIX.
2. Se invoca la función `pthread_getschedparam` para obtener los valores de los parámetros de planificación que en ese momento tiene el hilo, inicializando así las variables creadas anteriormente. Para invocar la función se debe indicar el identificador del hilo sobre del que se va a obtener la planificación, en este caso el propio hilo en ejecución, esto se consigue con una llamada a la función de POSIX `pthread_self`.
3. Se ajustan los parámetros de planificación deseados, en este caso una prioridad 3 y una planificación de tipo FIFO.
4. Finalmente se invoca la función `pthread_setschedparam` cuyo cometido es ajustar la planificación del hilo cuyo identificador recibe como primer parámetro con los valores que recibe como segundo y tercer parámetro.

El código descrito se sitúa al principio de cada una de las tres funciones que implementan las tareas permitiendo así que cada hilo que ejecute las funciones lo haga con la planificación correspondiente. Cabe destacar que las tres funciones ejecutan un bucle que se repite indefinidamente, por lo que si un hilo comienza a ejecutar una función siempre ejecutará esa función y no otra, con lo que se puede asegurar que todos los hilos ejecutarán con los parámetros de planificación correctos.

En este punto podemos asegurar que ninguno de los dos procesos que deben realizar su tarea de forma atómica será interrumpido. Sin embargo podemos asegurar que un proceso de prioridad inferior no pasará a ejecutar antes que uno de mayor prioridad en caso de conflicto sólo en el caso en que no existan bloqueos entre los procesos. En el sistema implementado sí que existen estos bloqueos, más concretamente en los mutex de la clase

`KFTaskManager` que sincronizan el acceso a los contenedores que almacenan las colas de datos y los estados de cada objeto.

El problema que se puede dar es conocido como “inversión de prioridades”. Este problema ocurre cuando existen varios procesos de diferente prioridad y los de baja prioridad pueden ser expulsados de la CPU. El problema podría suceder si se diese la siguiente situación:

1. Uno de los hilos que trata los datos consigue acceder al mutex que sincroniza el acceso contenedor que almacena el estado de los objetos, este contenedor es un recurso compartido con el proceso de inserción de datos, el de mayor prioridad.
2. Mientras el hilo de tratamiento está ejecutando la zona crítica se activa la ejecución del proceso de inserción de datos y en caso de que no haya más recursos disponibles se expulsa el proceso de tratamiento por tener menor prioridad y una planificación que permite la expulsión. Este proceso queda en estado de listo para ejecución.
3. Mientras el proceso de inserción ejecuta se activa el proceso de descarte. Este proceso quedará en estado de listo para ejecutar esperando, dado que el proceso de inserción tiene mayor prioridad que él y una planificación que no permite la expulsión.
4. El proceso de inserción, ahora en ejecución intenta obtener el mutex que sincroniza el acceso a los estados de los objeto y, dado que el mutex está bloqueado por el hilo de tratamiento el proceso de inserción debe bloquearse a la espera de poder acceder al mutex.
5. La situación ahora es que el hilo de mayor prioridad, el de inserción, está bloqueado esperando por el mutex que tiene el hilo de menor prioridad, el de tratamiento. Este hilo de tratamiento y el hilo de descarte, de prioridad mayor que el de tratamiento y menor que el de inserción están listos para ejecutar.
6. Ante esta situación el sistema operativo dará el uso del procesador al hilo de descarte hasta que este termine su ejecución y se bloqueé a la espera de su siguiente ciclo. Una vez que este hilo termine y se duerma entrará en ejecución el hilo de tratamiento ya que está en situación de listo para ejecución. Sólo cuando este hilo termine su ejecución en la zona crítica se desbloqueará el hilo de inserción y podrá ejecutar por tener mayor prioridad.

Como se puede observar se ha dado una situación en que el hilo de inserción, de mayor prioridad, es el último en ejecutar debido a que el bloqueo con el hilo de tratamiento permite que el hilo de descarte ejecute como si tuviese mayor prioridad que él, o lo que es lo mismo se ha producido una inversión de prioridad entre el hilo de inserción y el de descarte.

La solución a este problema se lleva a cabo aplicando un protocolo de prioridades dinámicas conocido como "protocolo de techo de prioridad". Este protocolo se basa en cambiar la prioridad de los hilos que acceden a cierto recurso, en este caso un mutex. Para aplicar el protocolo es necesario indicar al mutex cuál es su techo de prioridad, es decir, la prioridad que tiene el proceso de mayor prioridad que vaya acceder al mutex. Cuando un hilo accede al mutex su prioridad se modifica y se ajusta al techo de prioridad del mutex, o lo que es lo mismo la prioridad del proceso de mayor prioridad que pueda llegara a competir por el mutex, de esta forma sólo se evita la inversión de prioridades y los bloqueos sólo se producen mientras se ejecuten secciones críticas.

En este caso tenemos dos mutex que pueden ser fuente de conflicto, los mutex implementados en la clase `KFTaskManager`. La siguiente tabla muestra los techos de prioridad de ambos mutex y los hilos que involucran cada uno.

Mutex	Recurso	Hilos que acceden al mutex	Techo de prioridad
_m	Contenedor de estados de los objetos	• Inserción • Tratamiento	3 (Inserción)
_m2	Contenedor de colas de datos	• Inserción • Descarte	3 (Inserción)

Tabla 5.3. Techos de prioridad de los mutex de *KFTaskManager.*

Si repetimos el ejemplo anterior de inversión de prioridad pero haciendo uso del protocolo de techo de prioridad vemos que la inversión entre el hilo de inserción y el de descarte se elimina, ya que cuando el hilo de tratamiento toma el mutex `_m` aumenta su prioridad a la del techo de prioridad del mutex, es decir a prioridad 3, y por tanto terminará su ejecución en la zona crítica sin interrupciones por parte de otro hilos. Cuando el hilo de tratamiento termine de ejecutar su zona crítica su prioridad disminuirá a la que tenía anteriormente y por tanto los otros dos hilos que están esperando para ejecutar se disputarán el procesador y entrará a ejecutar el hilo de inserción por tener más prioridad que el de descarte. De esta forma se garantiza que el tiempo de bloqueo que sufren hilos de mayor prioridad por compartir recursos con hilos de menor prioridad es mínimo.

La forma de definir el techo de prioridad de los mutex es a través de la interfaz que ofrece POSIX, en la Figura 5.31 se muestra el código empleado para ajustar el techo de prioridad de los mutex.

```
template <class T>
KFTaskManager<T>::KFTaskManager():queuesPtr(),kfResults(),_m (),
_m2(){
      int priority = 3;

      pthread_mutexattr_t mAttr;
      pthread_mutexattr_init(&mAttr);

      pthread_mutexattr_setprotocol (&mAttr, PTHREAD_PRIO_PROTECT);
      pthread_mutexattr_setprioceiling(&mAttr, priority);

      pthread_mutex_init(_m.native_handle(), &mAttr);

      pthread_mutex_init(_m2.native_handle(), &mAttr);
     }
```

Figura 5.31. Código de configuración de techo de prioridad en mutex

La configuración del protocolo de techo de prioridad en los mutex de la clase `KFTaskManager` se lleva a cabo en el constructor de la clase, iniciando los mutex con el protocolo configurado.

Al comienzo del constructor se declara una variable que contiene el valor del techo de prioridad que tendrán los mutex, en este caso 3 como se indica en la Tabla 5.3. Posteriormente se declara una estructura de tipo `pthread_mutexattr_t`, esta estructura contiene todos los parámetros de configuración que tienen los mutex. Se inicializa la estructura anterior a través de la función `pthread_mutexattr_init`. Las dos siguientes líneas ejecutan las funciones que permiten indicar tanto el protocolo que se va a configurar, en este caso techo de prioridad, como el propio valor del techo de prioridad. Finalmente mediante la llamada a la función `pthread_mutex_init` se inicializan los mutex con los atributos previamente configurados.

5.9 PARALELIZACIÓN A NIVEL DE TAREA

Este último punto del desarrollo detalla el proceso seguido para realizar la paralelización del sistema a nivel de tarea. Durante esta fase se han llevado a cabo 4 implementaciones diferentes, utilizando diferentes técnicas, con el fin de analizar y valorar cada una de ellas.

Como se ha podido observar en puntos anteriores del desarrollo la mayor parte de los problemas que pueden surgir debido al uso de concurrencia en la aplicación, más concretamente, compartición de datos por varios hilos y condiciones de carrera han sido solventados al diseñar e implementar las clases que componen la aplicación pensando en la paralelización de las mismas, aunque en este punto se detallaran como se han solventado estos problemas.

5.9.1 PARALELIZACIÓN CON HILOS NATIVOS DE C++

La primera técnica que se ha utilizado para la paralelización a nivel de tarea del sistema está basada en hilos nativos de C++. Como ya se ha indicado en el punto 2.8.7 C++11 incluye soporte para la creación de aplicaciones basadas en hilos, por lo que se ha querido aprovechar estas facilidades.

La implementación con hilos nativos está compuesta de dos funciones, la primera de ellas, `nativeThreads`, se encarga de crear inicializar el sistema y crear los hilos, la segunda, `tratar`, es la función que ejecutan los hilos y se encarga de obtener los datos a tratar y aplicarles el algoritmo del filtro de Kalman, además en los casos de los algoritmos de descarte II y III también efectúa tareas de descarte como se explicó en el punto 0 de este documento.

Por otra parte existen otras dos funciones, manejadas por otros dos hilos independientes que se encargan de llevar a cabo la inserción de datos y, en los casos en que se requiere, tareas de descarte, como en el algoritmo I y III así como toma de datos para la realización de las pruebas. El punto se centrará en la función `nativeThreads` y la función `tratarNat`.

5.9.1.1 FUNCIÓN NATIVETHREADS

La función `nativeThreads` es la encargada inicializar todo el sistema y crear los hilos que posteriormente, tratarán los datos. La función recibe dos parámetros, el primero de ellos

es un entero que indica el número de hilos que deberá crear para tratar los datos, el segundo es una cadena que indica la ruta donde se encuentra el fichero con los datos simulados.

En esta función se crean 3 estructuras:

- Vector de objetos `thread`: Esta estructura contiene los objetos que encapsulan los hilos, de forma que siempre se tiene una referencia a ellos.
- Instancia de la clase `DataLoader`: Como se ha explicado anteriormente, esta clase auxiliar es la encargada de llevar a cabo varias funciones con los datos, más concretemente cargarlos desde disco a memoria y posteriormente insertarlos en las colas simulando su llegada en el momento correcto.
- Instancia de la clase `KFTaskManager`: Esta clase permite gestionar tanto la llegada de tareas al sistema, como la obtención de las mismas por parte de los hilos.

La primera acción que se lleva a cabo es invocar a la función `start` de la clase `DataLoader` a través de su instancia. Esta función recibe como parámetro la ruta absoluta donde se encuentra el fichero de datos que se pretende cargar, carga los datos en memoria ordenados y retorna una referencia a un objeto de tipo `KFTaskManager`. Esta referencia será la que se use para la ejecución del sistema, ya que tanto la clase `DataLoader` como la función de tratamiento de datos, que posteriormente se explicará, deben compartir la misma instancia, una para poder insertar los datos en las colas y otra para poder extraerlos y tratarlos.

Una vez que los datos de las medidas han sido cargados en memoria se crean los hilos que manejarán las funciones auxiliares encargadas de la introducción de datos en las colas y de, dependiendo del algoritmo que se utilice, el descarte de datos.

Finalmente se crean los hilos que se encargarán de tratar los datos. Esto se hace mediante un bucle cuyo número de iteraciones está determinado por el número de hilos que indiquen los parámetros que recibe la función. Cada uno de estos hilos, encapsulado por un objeto `thread` de la biblioteca estándar de C++11, se almacena en el vector creado anteriormente para tal efecto. Los hilos creados ejecutarán la función `tratarNat`, la cual se describirá a continuación. Por último en otro bucle se hará invocará a la función `join` de todos los hilos contenidos en el bucle. Esta función hace que el hilo principal espere a que los demás hilos terminen antes de continuar.

5.9.1.2 FUNCIÓN TRATARNAT

La función `tratarNat` implementa las acciones necesarias para llevar a cabo la aplicación del algoritmo de Kalman sobre los datos que esperan en las colas. En el caso del algoritmo II implementa el descarte y en el algoritmo III parte de él.

Esta función recibe como parámetro una referencia al objeto `KFTaskManager` declarado en la función `nativeThreads`. Este objeto encapsula las colas de datos y toda la lógica necesaria para acceder a ellos.

Las acciones se ejecutan constantemente en un bucle infinito y varían según el algoritmo de descarte utilizado:

5.9.1.3 ALGORITMO I

En el primer algoritmo de descarte la función de tratamiento de datos se dedica exclusivamente al tratamiento de datos, su funcionamiento es el siguiente.

- Se obtiene un puntero a una cola a través de la llamada a la función `getTask` de la clase `KFTaskManager`. Como se ha explicado anteriormente esta acción extrae el puntero de la cola circular y por tanto ningún otro hilo podrá acceder a esos mismos datos hasta que no se vuelva a insertar.
- Si la cola que se ha obtenido está vacía, se vuelve a insertar el puntero en la cola circular mediante la función `endTask` de la clase `KFTaskManager` a través del objeto de esta clase que se recibe por parámetro y se vuelve al principio. En este momento la cola está a disposición de cualquier otro hilo ya que el puntero de la referencia se reinserta en la cola circular.
- En caso de que la cola no esté vacía, se extrae el primer dato de la misma, esto se consigue mediante la llamada a la función `poll` de la clase `KFQueue`, de la cual es instancia la cola en cuestión.
- Se obtiene el un puntero al objeto de tipo `KFLib` asociado al objeto móvil del cual se va tratar el dato a mediante de la llamada a la función `getKFLib` de la clase `KFTaskmanager`.
- Se aplica el algoritmo sobre los datos invocando a las funciones pertinentes de la clase `KFLib` a través del puntero a una instancia de esta clase que se ha recuperado en el paso anterior.

- Finalizada la aplicación del algoritmo se reintroduce el puntero a la cola de datos en la cola circular de donde se extrajo, permitiendo a cualquier otro hilo acceder a él y por tanto a esa cola y esos datos. Termina la iteración del bucle y se vuelve a repetir el proceso indefinidamente.

Este algoritmo se representa en el diagrama siguiente:

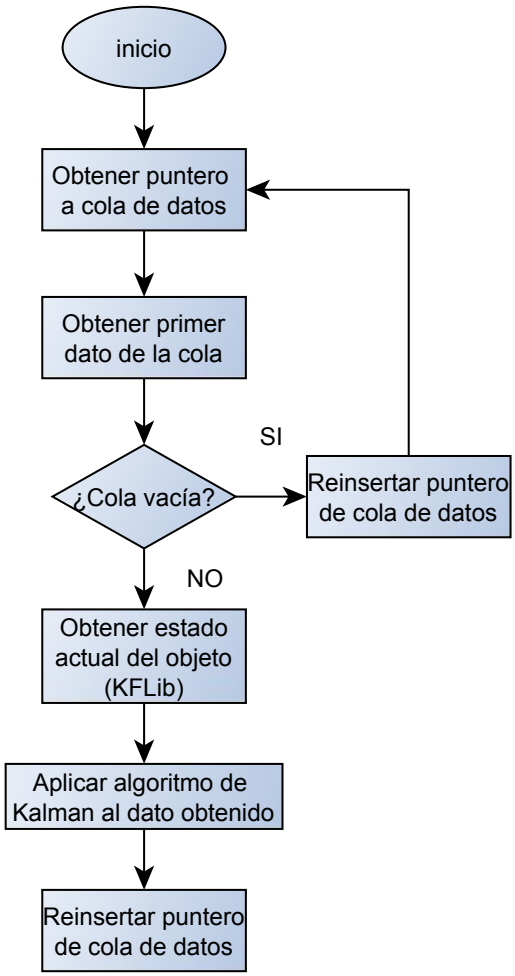

Figura 5.32. Diagrama de la función *tratarNat*

5.9.1.4 ALGORITMO II

En el caso de este algoritmo de descarte la función `tratarNat` también se ocupa de eliminar de las colas los datos que no se pueden tratar, su funcionamiento es el siguiente:

- Se obtiene un puntero a una cola a través de la llamada a la función `getTask` de la clase `KFTaskManager`.

- Si la cola que se ha obtenido está vacía, se vuelve a insertar el puntero en la cola circular mediante la función `endTask` de la clase `KFTaskManager` a través del objeto de esta clase que se recibe por parámetro y se vuelve al principio. En este momento la cola está a disposición de cualquier otro hilo ya que el puntero que la referencia se reinserta en la cola circular.
- En caso de que la cola no esté vacía, se extrae el primer dato de la misma, esto se consigue mediante la llamada a la función `poll` de la clase `KFQueue`, de la cual es instancia la cola en cuestión.
- Se toma el tiempo del sistema en ese instante. A ese tiempo tomado se le resta el tiempo en que se empezó a ejecutar, es decir, se calcula cuanto tiempo lleva el sistema en ejecución.
- Se compara el resultado del cálculo anterior con la marca de tiempo, de esta manera vemos si el sistema está tratando datos con una marca de tiempo mucho anterior al instante actual y por lo tanto podemos cuantificar el retraso del sistema. Dependiendo de este retraso se aplicará un descarte como el indicado en la Tabla 5.1.
- Se obtiene el un puntero al objeto de tipo `KFLib` asociado al objeto móvil del cual se va tratar el dato que se ha obtenido previamente mediante de la llamada a la función `getKFLib` de la clase `KFTaskmanager`.
- Se aplica el algoritmo sobre los datos invocando a las funciones pertinentes de la clase `KFLib` a través del puntero a una instancia de esta clase que se ha recuperado en el paso anterior.
- Dependiendo del retraso calculado anteriormente se eliminan los elementos correspondientes de la cola con la que se está trabajando.
- Finalizada la aplicación del algoritmo y el descarte correspondiente se reintroduce el puntero a la cola de datos en la cola circular de donde se extrajo, permitiendo a cualquier otro hilo acceder a él y por tanto a esa cola y esos datos. Termina la iteración del bucle y se vuelve a repetir el proceso indefinidamente.

Este procedimiento se representa en la siguiente figura:

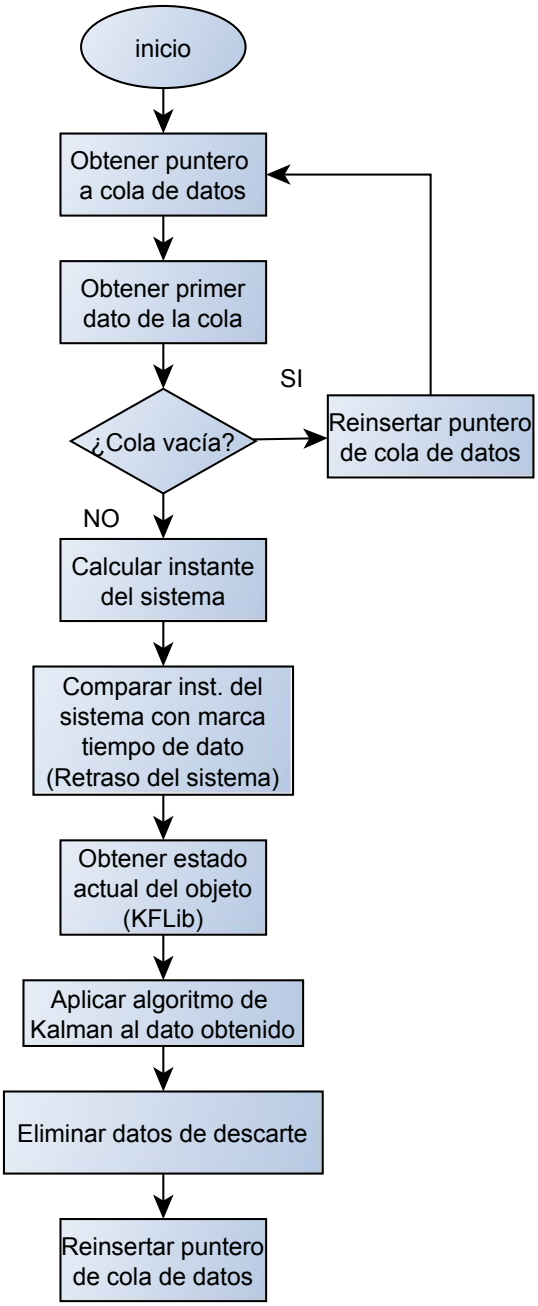

Figura 5.33. Diagrama de la función *tratarNat* para el algoritmo de descarte II

5.9.1.5 ALGORITMO III

Para el caso del tercer algoritmo la función `tratarNat` se hace cargo de parte del descarte, más concretamente de llevar a cabo los recálculos del factor de descarte y de eliminar los datos descartados, su funcionamiento es el siguiente:

- Se obtiene un puntero a una cola a través de la llamada a la función `getTask` de la clase `KFTaskManager`. Como se ha explicado anteriormente esta acción extrae el puntero de la cola circular y por tanto ningún otro hilo podrá acceder a esos mismos datos hasta que no se vuelva a insertar.
- Si la cola que se ha obtenido está vacía, se vuelve a insertar el puntero en la cola circular mediante la función `endTask` de la clase `KFTaskManager` a través del objeto de esta clase que se recibe por parámetro y se vuelve al principio. En este momento la cola está a disposición de cualquier otro hilo ya que el puntero que la referencia se reinserta en la cola circular.
- En caso de que la cola no esté vacía, se extrae el primer dato de la misma, esto se consigue mediante la llamada a la función `poll` de la clase `KFQueue`, de la cual es instancia la cola en cuestión.
- Se comprueba si el valor del factor de descarte del objeto es -1. Si es así se recalcula el valor de descarte del objeto como se ha explicado en el punto 5.6.3 de este documento.
- Se obtiene el un puntero al objeto de tipo `KFLib` asociado al objeto móvil del cual se va tratar el dato a mediante de la llamada a la función `getKFLib` de la clase `KFTaskmanager`.
- Se aplica el algoritmo sobre los datos invocando a las funciones pertinentes de la clase `KFLib` a través del puntero a una instancia de esta clase que se ha recuperado en el paso anterior.
- Se eliminan de la cola de datos en uso tantos elementos como indique el factor de descarte del objeto.
- Finalizada la aplicación del algoritmo y del descarte se reintroduce el puntero a la cola de datos en la cola circular de donde se extrajo, permitiendo a cualquier otro hilo acceder a él y por tanto a esa cola y esos

datos. Termina la iteración del bucle y se vuelve a repetir el proceso indefinidamente.

Este procedimiento se representa en la siguiente figura:

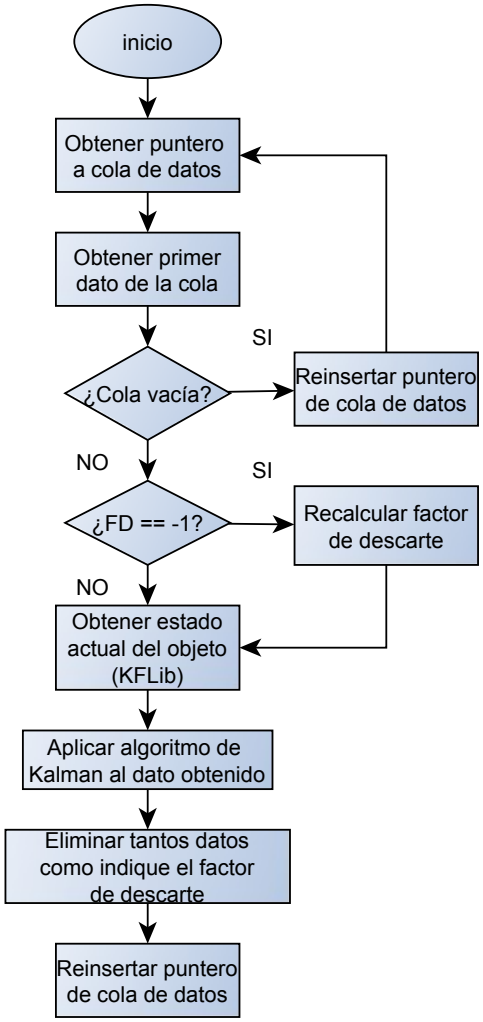

Figura 5.34. Diagrama de la función *tratarNat* para el algoritmo de descarte III

5.9.2 PARALELIZACIÓN CON OBJETOS *TASK* DE TBB

TBB es una librería de paralelización que ofrece numerosos mecanismos. Uno de estos mecanismos es la paralelización de tareas. Esta forma de paralelización se basa en definir tareas a las que posteriormente el planificador de TBB asignará un hilo y ejecutará.

La principal ventaja que aporta sobre la técnica anterior es que abstrae al programador de la creación, planificación y gestión de hilos, ya que es la propia librería la que se encarga de crear el número de hilos que considere idóneo para la máquina donde se ejecuta, así como de gestionar su uso y posterior destrucción.

Para paralelizar tareas con TBB es necesario definir qué es una tarea, en este caso, una tarea es, "la aplicación del algoritmo del filtro de Kalman para cada dato de medida de un objeto móvil que llega al sistema". Con esta definición se debe crear un objeto que la represente en el sistema y que tenga la capacidad de llevar a cabo todas las operaciones que conlleva su realización, así como gestionar los datos que maneje.

La representación de las tareas se hace a través de una clase o estructura que hereda de la clase `task` de TBB. En este caso se ha optado por definir una estructura ya que resulta más simple.

```
struct KFTaskLst:public task{
 public:
   KFTaskLst(KFTaskManager<double>& tm, KFData<double> k,
   KFQueue<double>* queue):_tm(tm), _k(k), _queue(queue){}

   task* execute(){
      //!Tratamiento de datos
      KFLib<double>* klib;
      klib = this->_tm.getKFLib(this->_k.id());
      klib->setH(this->_k.hasVel());
      klib->updateR(this->_k.covXX(), this->_k.covYY());
      klib->correct(this->_k.kf());
      klib->predict();
      this->_tm.endTask(this->_queue);
      return nullptr;
   }

 private:

      KFTaskManager<double> & _tm;
      KFData<double> _k;
      KFQueue<double>* _queue;
};
```

Figura 5.35. Estructura que representa una tarea.

Esta estructura, al heredar de la clase `task` debe implementar una función cuyo prototipo es `task* execute()`. Esta función será la que ejecute el hilo al que se le asigne la tarea, por lo que debe implementar toda la funcionalidad para ejecutar la tarea completa. Por otra parte, como se puede observar en el prototipo de la función, no acepta ningún parámetro. Para solventar este inconveniente se utilizan los atributos de la estructura.

En la Figura 5.35 se detalla la implementación de la estructura que define una tarea. Como se puede observar la estructura tiene 3 atributos miembro:

- Referencia a tipo `KFTaskManager`: Esta referencia permite acceder a los métodos de esta clase así como a los datos que encapsula, más concretamente es necesaria para obtener el objeto `KFLib` que representa el estado del objeto, y para acceder a la cola circular de punteros cuando se termine de realizar la tarea.
- Objeto tipo `KFData`: Este objeto contiene los datos de la medida.
- Puntero de tipo `KFQueue`: Es el puntero a la cola de datos del objeto que se va a tratar. Es necesario almacenarlo ya que al finalizar la tarea hay que devolverlo a la cola circular para que vuelva a estar disponible.

Estos atributos se inicializan al crear la tarea, posteriormente, cuando la tarea sea ejecutada por algún hilo del sistema, este ejecutará la función `execute`, desde esta función se puede acceder a los atributos de la estructura y por tanto a los datos necesarios para ejecutar la tarea sin necesidad de pasarlos como parámetro a la función.

La función `execute` retorna un puntero a un objeto tipo `task`, esto se hace muy útil cuando dentro de la tarea se genera otra, en este caso se puede retornar el puntero a la tarea generada para que entre en ejecución y aproveche los recursos que ha dejado libre su predecesora, optimizando así la reserva y liberación de recursos.

Cada vez que se crea una tarea, es necesario ponerla a disposición del planificador de TBB para su ejecución cuando existan recursos. Esto se consigue con la función `spawn(task& t)`. Esta función pone la tarea en un *pool* de tareas listas para ejecución y retorna inmediatamente, con lo que el planificador de TBB en cualquier momento puede decidir ejecutarla.

La implementación de este tipo de paralelización se lleva a cabo en la función llamada `TaskPar`, esta función recibe 2 parámetros, el primero es un entero que indica el número de

hilos que se recomienda al planificador usar, al igual que en el caso anterior si este valor es 0 la librería de TBB decidirá cuál es el número optimo de hilos dependiendo de la máquina donde se ejecute. El segundo parámetro indica el nombre del fichero con los datos de la simulación.

Al comienzo de la función se ajustan los parámetros de planificación de los hilos. Posteriormente se inicializa el planificador de tareas de TBB. La inicialización se hace con un objeto tipo `task_scheduler_init`. Este objeto tienen dos constructores, el primero de ellos es un constructor por defecto, con la llamada a este constructor se inicializa el planificador y este se encarga de decidir cuál es el número óptimo de hilos que debe crear dependiendo de las características del hardware del sistema. El segundo de los constructores recibe un parámetro, en forma de un entero sin signo, que indica el número de hilos que debe crear el planificador para trabajar. Con este constructor el programador puede "aconsejar" al planificador el número de hilos con que quiere ejecutar la aplicación.

Una vez que se ha inicializado y configurado el planificador de tareas, se pasa a cargar los datos de fichero e inicializar el sistema gestor de tareas mediante la llamada a la función `start` de la clase `DataLoader` como se ha explicado anteriormente en el documento.

Por último la función entra en un bucle donde obtendrá los datos de las colas correspondientes, generará las tareas y las pondrá a disposición del planificador de TBB para su posterior ejecución. Dependiendo del algoritmo de descarte que se utilice cambiarán las acciones que se realicen dentro de este bucle, a continuación se detallan por algoritmo.

5.9.2.1 ALGORITMO I

En el caso del primer algoritmo de descarte, dentro del bucle simplemente se lleva a cabo la obtención de datos de las colas y su posterior conversión en tareas de TBB para su ejecución. Las acciones que se realizan son las siguientes:

- Primeramente se obtiene un puntero a una de las colas de datos del sistema con la llamada a la función `getTask`.
- Se comprueba que esta cola tenga datos, en caso contrario se reinserta el puntero obtenido anteriormente a través de la función `endTask` y se vuelve al comienzo del bucle.
- En caso de que la cola tenga datos, se extrae el primer dato de la cola llamando a la función `poll` de la clase `KFQueue` a través del puntero de esta clase obtenido anteriormente.

- Una vez que se ha obtenido el puntero a la cola y su primer dato, se tiene toda la información necesaria para generar una tarea. Para ello se crea un nuevo objeto de tipo `KFTaskLst`, cuyo constructor recibe como parámetros una referencia al objeto `KFTaskManager` creado al principio de la función, el puntero a la cola de datos con la que se está trabajando y los datos que se han extraído de la cola, estos último se pasan utilizando semántica de movimiento para minimizar las operaciones de memoria. La creación de este objeto debe hacerse con el operador `new` para crearlo en memoria dinámica, ya que en otro caso se destruiría al terminar la iteración del bucle por quedar fuera de ámbito.
- Finalmente, una vez creada la tarea se invoca el método `tbb::task::spawn` y se le pasa como parámetro una referencia a la tarea recién creada. A partir de este momento la tarea queda lista para su ejecución cuando el planificador lo decida. El bucle termina y vuelve al primer paso.

Cabe destacar que aunque la tarea se haya generado en memoria dinámica utilizando el operador `new` TBB se encarga de liberar esa memoria al finalizar la tarea. El algoritmo se representa en la siguiente figura.

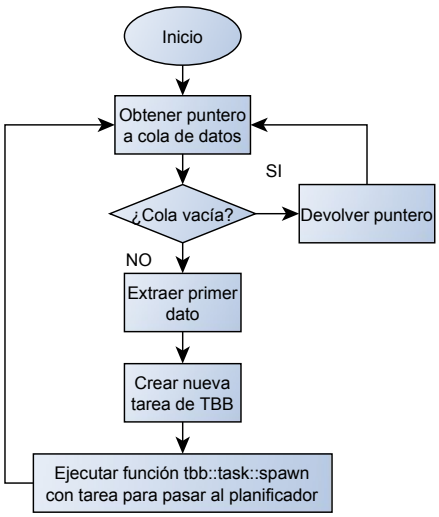

Figura 5.36. Diagrama de generación y ejecución de tareas con algoritmo de descarte I

5.9.2.2 ALGORITMO II

En el caso del algoritmo II el bucle, además de obtener los datos y generar las tares de TBB, se encarga del descarte, las acciones que realiza son las siguientes:

- Primeramente se obtiene un puntero a una de las colas de datos del sistema con la llamada a la función `getTask`.
- Se comprueba que esta cola tenga datos, en caso contrario se reinserta el puntero obtenido anteriormente a través de la función `endTask` y se vuelve al comienzo del bucle.
- En caso de que la cola tenga datos, se extrae el primer dato de la cola llamando a la función `poll` de la clase `KFQueue` a través del puntero de esta clase obtenido anteriormente.
- Se toma el tiempo del sistema en ese instante. A ese tiempo tomado se le resta el tiempo en que se empezó a ejecutar, es decir, se calcula cuanto tiempo lleva el sistema en ejecución.
- Se compara el resultado del cálculo anterior con la marca de tiempo, de esta manera vemos si el sistema está tratando datos con una marca de tiempo mucho anterior al instante actual y por lo tanto podemos cuantificar el retraso del sistema. Dependiendo de este retraso se aplicará un descarte como el indicado en la Tabla 5.1
- Una vez que se ha obtenido el puntero a la cola y su primer dato, se tiene toda la información necesaria para generar una tarea. Para ello se crea un nuevo objeto de tipo `KFTaskLst`, cuyo constructor recibe como parámetros una referencia al objeto `KFTaskManager` creado al principio de la función, el puntero a la cola de datos con la que se está trabajando y los datos que se han extraído de la cola, estos últimos se pasan utilizando semántica de movimiento para minimizar las operaciones de memoria. La creación de este objeto debe hacerse con el operador new para crearlo en memoria dinámica, ya que en otro caso se destruiría al terminar la iteración del bucle por quedar fuera de ámbito.
- Dependiendo del retraso calculado anteriormente se eliminan los elementos correspondientes de la cola con la que se está trabajando.

- Finalmente, una vez creada la tarea se invoca el método `tbb::task::spawn` y se le pasa como parámetro una referencia a la tarea recién creada. El bucle termina y vuelve al primer paso.

La siguiente figura muestra el algoritmo utilizado.

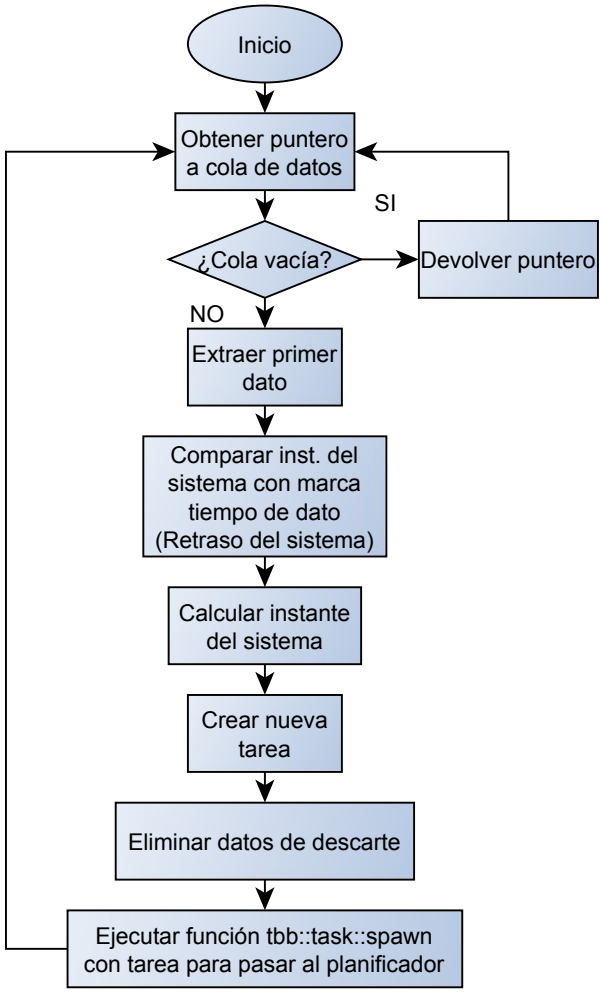

Figura 5.37. Diagrama de generación y ejecución de tareas con algoritmo de descarte II.

5.9.2.3 ALGORITMO III

En el caso del algoritmo III, el bucle, además de generar las tareas de TBB lleva a cabo el cálculo del factor de descarte y la eliminación de los datos sobrantes.

- Se obtiene un puntero a una cola a través de la llamada a la función `getTask` de la clase `KFTaskManager`. Como se ha explicado anteriormente esta acción extrae el puntero de la cola circular y por tanto ningún otro hilo podrá acceder a esos mismos datos hasta que no se vuelva a insertar.
- Si la cola que se ha obtenido está vacía, se vuelve a insertar el puntero en la cola circular mediante la función `endTask` de la clase `KFTaskManager`.
- En caso de que la cola no esté vacía, se extrae el primer dato de la misma, esto se consigue mediante la llamada a la función `poll` de la clase `KFQueue`, de la cual es instancia la cola en cuestión.
- Se comprueba si el valor del factor de descarte del objeto es -1. Si es así se recalcula el valor de descarte del objeto como se ha explicado en el punto 5.6.3 de este documento.
- Una vez que se ha obtenido el puntero a la cola y su primer dato, se tiene toda la información necesaria para generar una tarea. Para ello se crea un nuevo objeto de tipo `KFTaskLst`, cuyo constructor recibe como parámetros una referencia al objeto `KFTaskManager` creado al principio de la función, el puntero a la cola de datos con la que se está trabajando y los datos que se han extraído de la cola, estos últimos se pasan utilizando semántica de movimiento para minimizar las operaciones de memoria. La creación de este objeto debe hacerse con el operador `new` para crearlo en memoria dinámica, ya que en otro caso se destruiría al terminar la iteración del bucle por quedar fuera de ámbito.
- Se eliminan de la cola de datos en uso tantos elementos como indique el factor de descarte del objeto.
- Finalmente, una vez creada la tarea se invoca el método `tbb::task::spawn` y se le pasa como parámetro una referencia a la tarea recién creada. El bucle termina y vuelve al primer paso.

La siguiente figura muestra el diagrama del proceso explicado.

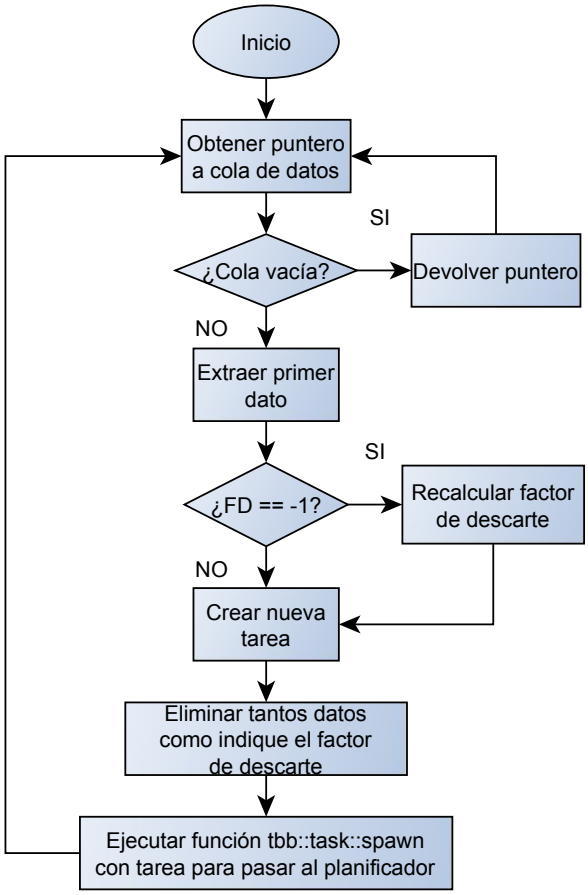

Figura 5.38. Diagrama de generación y ejecución de tareas con algoritmo de descarte III.

5.9.3 PARALELIZACIÓN CON TASKENQUEUE DE TBB

Este método de paralelización difiere del anterior en que en este las tareas que se van generando se encolan para su posterior ejecución. Al igual que en el caso anterior es necesario definir las tareas, para ello se utiliza la misma estructura que en el caso anterior, representada en la Figura 5.35, aunque en este caso su nombre cambia a `KFTaskEnq`.

La función `tbb::task::enqueue` permite encolar las tareas que se generan en una cola interna del planificador, este a su vez, según vaya teniendo recursos libres irá seleccionando tareas para ejecutarlas. La forma de seleccionarlas es en orden de llegada, es decir, la cola del planificador es una estructura FIFO.

El funcionamiento es el siguiente:

- Se obtiene un puntero a una cola de datos.
- Se crea una tarea a partir del primer elemento almacenado en la cola de datos.
- La tarea creada se encola en el planificador a través del método `tbb::task::enqueue`.
- El planificador toma la tarea cuando tenga recursos y la ejecuta.
- Este proceso se repite indefinidamente.

Como se puede observar el mecanismo es muy sencillo para el programador que solo tiene que crear tareas según vaya necesitándolas y poniéndolas en cola, posteriormente es la propia librería la que se encarga de todo.

La implementación de este tipo de paralelización se ha llevado a cabo en una función llamada `taskEnqueue`, esta función recibe 2 parámetros, el primero es un entero que indica el número de hilos que se usarán, al igual que en el caso anterior si este valor es 0 la librería de TBB decidirá cuál es el número optimo de hilos dependiendo de la máquina donde se ejecute. El segundo parámetro indica el nombre del fichero con los datos de la simulación.

Al comienzo de la función se ajustan los parámetros de prioridad y planificación de los hilos como se ha explicado anteriormente. Posteriormente se inicializa el planificador de TBB, inicializando un objeto `task_scheduler_init`, en caso de que el primer parámetro que recibe la función sea distinto de 0 se pasa este valor como parámetro al constructor del planificador, obligando a este a iniciar un número concreto de hilos. En caso contrario no se le

pasa ningún valor al constructor, iniciando él mismo el número de hilos que considere óptimos respecto de las características del sistema.

A continuación, al igual que en los métodos anteriores se crean los objetos `DataLoader` y `KFTaskManager` y se cargan los datos desde el fichero recibido por parámetro a memoria.

Tras cargar los datos en memoria se lanzan los hilos que ejecutarán funciones auxiliares, como el caso de la función de inserción de datos y dependiendo del algoritmo de descarte que utilice la función de descarte de los mismos.

Finalmente la función entra en un bucle infinito donde se crean y encolan las tareas. El contenido de este bucle difiere dependiendo del algoritmo de descarte que utilice.

5.9.3.1 ALGORITMO I

En el caso de este algoritmo dentro del bucle solo se lleva a cabo la creación de tareas y el encolado.

- Primeramente se obtiene un puntero a una de las colas de datos del sistema con la llamada a la función `getTask`.
- Se comprueba que esta cola tenga datos, en caso contrario se reinserta el puntero obtenido anteriormente a través de la función `endTask` y se vuelve al comienzo del bucle.
- En caso de que la cola tenga datos, se extrae el primer dato de la cola llamando a la función `poll` de la clase `KFQueue` a través del puntero de esta clase obtenido anteriormente.
- Una vez que se ha obtenido el puntero a la cola y su primer dato, se tiene toda la información necesaria para generar una tarea. Para ello se crea un nuevo objeto de tipo `KFTaskEnq`, cuyo constructor recibe como parámetros una referencia al objeto `KFTaskManager` creado al principio de la función, el puntero a la cola de datos con la que se está trabajando y los datos que se han extraído de la cola, estos último se pasan utilizando semántica de movimiento para minimizar las operaciones de memoria. La creación de este objeto debe hacerse con el operador `new` para crearlo en memoria dinámica, ya que en otro caso se destruiría al terminar la iteración del bucle por quedar fuera de ámbito.

- Finalmente, una vez creada la tarea se invoca el método `tbb::task::enqueue` y se le pasa como parámetro una referencia a la tarea recién creada. El bucle termina y vuelve al primer paso.

Cabe destacar que aunque la tarea se haya generado en memoria dinámica utilizando el operador `new` TBB se encarga de liberar esa memoria al finalizar la tarea. El algoritmo se representa en la siguiente figura.

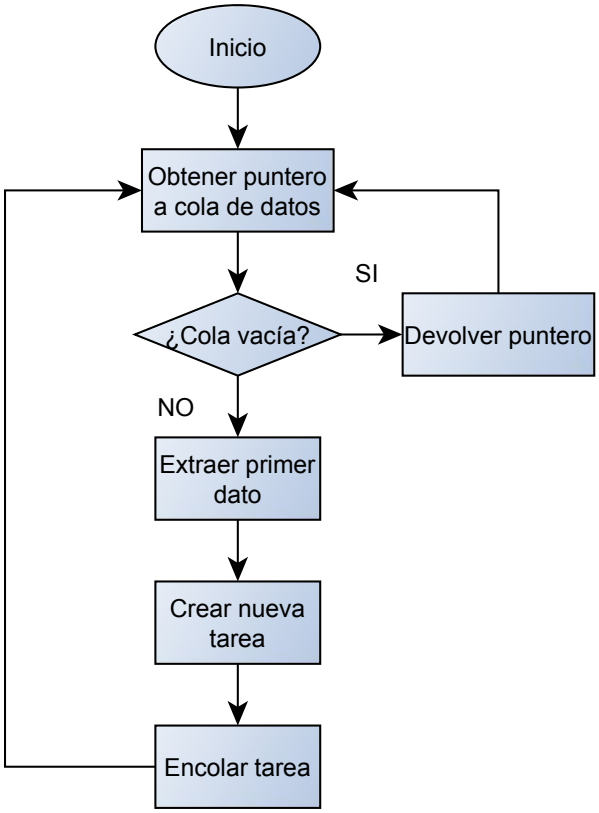

Figura 5.39. Esquema de creación y encolado de tareas con algoritmo I de descarte

5.9.3.2 ALGORITMO II

En este caso el bucle se encarga, además de crear las tareas, de llevar a cabo el descarte, los criterios de descarte son los mismos que los expuestos en la Tabla 5.1.

- Primeramente se obtiene un puntero a una de las colas de datos del sistema con la llamada a la función `getTask`.
- Se comprueba que esta cola tenga datos, en caso contrario se reinserta el puntero obtenido anteriormente a través de la función `endTask` y se vuelve al comienzo del bucle.
- En caso de que la cola tenga datos, se extrae el primer dato de la cola llamando a la función `poll` de la clase `KFQueue` a través del puntero de esta clase obtenido anteriormente.
- Se toma el tiempo del sistema en ese instante. A ese tiempo tomado se le resta el tiempo en que se empezó a ejecutar, es decir, se calcula cuanto tiempo lleva el sistema en ejecución.
- Se compara el resultado del cálculo anterior con la marca de tiempo, de esta manera vemos si el sistema está tratando datos con una marca de tiempo mucho anterior al instante actual y por lo tanto podemos cuantificar el retraso del sistema. Dependiendo de este retraso se aplicará un descarte como el indicado en la Tabla 5.1
- Una vez que se ha obtenido el puntero a la cola y su primer dato, se tiene toda la información necesaria para generar una tarea. Para ello se crea un nuevo objeto de tipo `KFTaskEnq`, cuyo constructor recibe como parámetros una referencia al objeto `KFTaskManager` creado al principio de la función, el puntero a la cola de datos con la que se está trabajando y los datos que se han extraído de la cola, estos últimos se pasan utilizando semántica de movimiento para minimizar las operaciones de memoria. La creación de este objeto debe hacerse con el operador new para crearlo en memoria dinámica, ya que en otro caso se destruiría al terminar la iteración del bucle por quedar fuera de ámbito.
- Dependiendo del retraso calculado anteriormente se eliminan los elementos correspondientes de la cola con la que se está trabajando.

- Finalmente, una vez creada la tarea se invoca el método `tbb::task::enqueue` y se le pasa como parámetro una referencia a la tarea recién creada. El bucle termina y vuelve al primer paso.

La siguiente figura representa el proceso explicado.

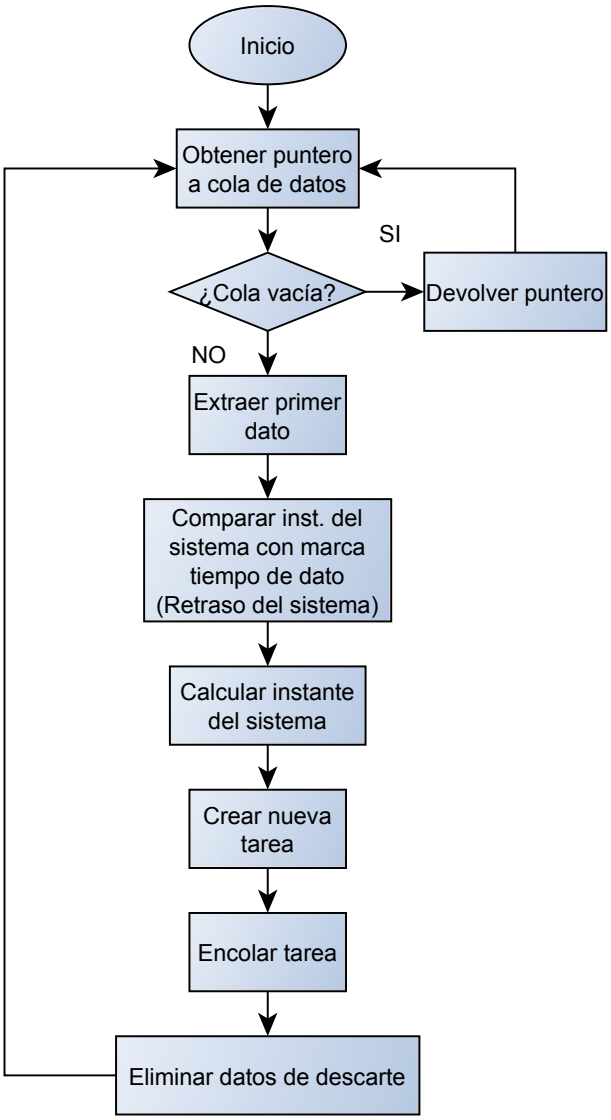

Figura 5.40. Esquema de creación de tareas con algoritmo II de descarte

5.9.3.3 ALGORITMO III

Para el caso del algoritmo III el bucle, además de crear las tareas, también parte en el descarte de datos.

- Se obtiene un puntero a una cola a través de la llamada a la función `getTask` de la clase `KFTaskManager`. Como se ha explicado anteriormente esta acción extrae el puntero de la cola circular y por tanto ningún otro hilo podrá acceder a esos mismos datos hasta que no se vuelva a insertar.
- Si la cola que se ha obtenido está vacía, se vuelve a insertar el puntero en la cola circular mediante la función `endTask` de la clase `KFTaskManager`.
- En caso de que la cola no esté vacía, se extrae el primer dato de la misma, esto se consigue mediante la llamada a la función `poll` de la clase `KFQueue`, de la cual es instancia la cola en cuestión.
- Se comprueba si el valor del factor de descarte del objeto es -1. Si es así se recalcula el valor de descarte del objeto como se ha explicado en el punto 5.6.3 de este documento.
- Una vez que se ha obtenido el puntero a la cola y su primer dato, se tiene toda la información necesaria para generar una tarea. Para ello se crea un nuevo objeto de tipo `KFTaskEnq`, cuyo constructor recibe como parámetros una referencia al objeto `KFTaskManager` creado al principio de la función, el puntero a la cola de datos con la que se está trabajando y los datos que se han extraído de la cola, estos últimos se pasan utilizando semántica de movimiento para minimizar las operaciones de memoria. La creación de este objeto debe hacerse con el operador new para crearlo en memoria dinámica, ya que en otro caso se destruiría al terminar la iteración del bucle por quedar fuera de ámbito.
- Finalmente, una vez creada la tarea se invoca el método t`bb::task::enqueue` y se le pasa como parámetro una referencia a la tarea recién creada. El bucle termina y vuelve al primer paso.
- Se eliminan de la cola de datos en uso tantos elementos como indique el factor de descarte del objeto.

La siguiente figura muestra el diagrama del proceso descrito.

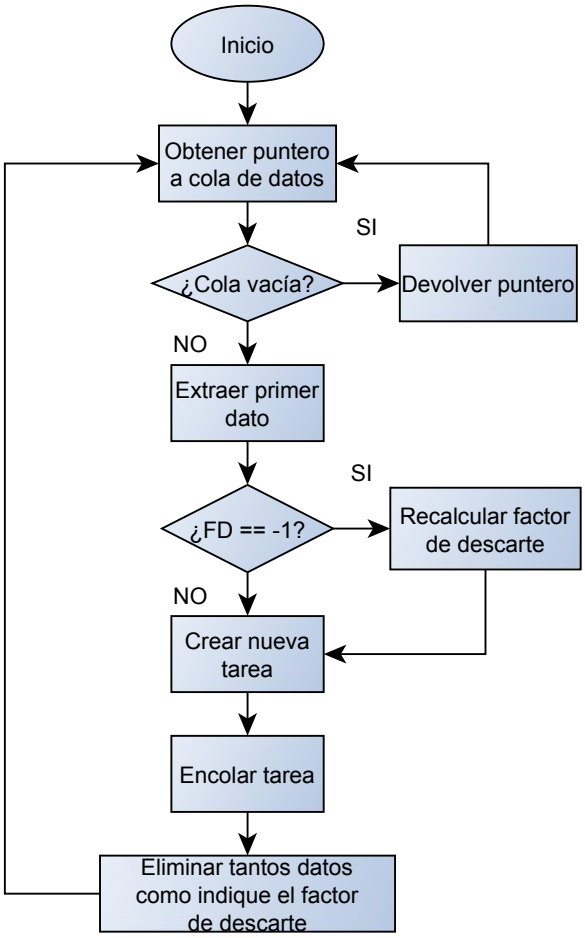

Figura 5.41. Esquema de creación de tareas con algoritmo III de descarte

5.9.4 PARALELIZACIÓN CON *PIPELINE* DE TBB

La última forma de paralelización que se ha implementado se basa en segmentar las tareas en subtareas y ejecutar estas en diferentes etapas. De esta forma se consigue tanto paralelismo a nivel de tareas como a nivel de datos, ya que permite que varios datos puedan estar en diferentes etapas produciendo un tratamiento más eficiente de los mismos.

Un *pipeline*, como ya se ha explicado en el capítulo 2 de este documento, es una "tubería" por la que fluyen los datos y, a lo largo de ella, se les aplican operaciones. Estas operaciones se aplican en los diferentes segmentos de la tubería, a los que se conoce como filtros o etapas. Cada conjunto de operaciones que se aplica en una etapa se llama subtarea y surge de dividir las operaciones que componen una tarea en tantos grupos como etapas tiene la tubería o *pipeline*.

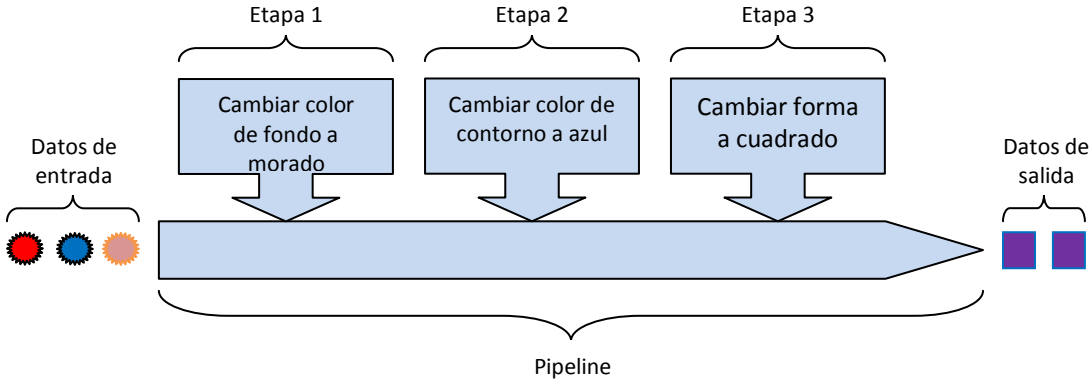

Figura 5.42. Representación de un *pipeline*

La Figura 5.42 muestra de manera gráfica un *pipeline* en cuyo interior se realizan transformaciones sobre figuras geométricas, estas figuras geométricas representan los datos de entrada del p*ipeline.* Como se puede observar, los datos recorren el *pipeline* de izquierda a derecha, ya que el recorrido es un solo sentido y no es posible retroceder. A lo largo de su recorrido cada figura va pasando por las distintas fases o etapas que componen el *pipeline.* Cada una de estas etapas aplica una serie de operaciones sobre los datos. El paso por cada etapa se realiza de forma secuencial, es decir, un dato no puede pasar a una etapa si aún no ha terminado la anterior, tampoco se puede modificar el orden de las etapas.

La salida de datos se realiza en el mismo orden en que entran siempre y cuando cada etapa sea ejecutada por un solo hilo.

Con un *pipeline* en el que cada etapa esté ejecutada por un hilo se consigue paralelismo de datos, ya que a cada dato se le hace pasar por un flujo de tareas, consiguiéndose aplicar las operaciones sobre varios datos al mismo tiempo. Para sacar más partido al *pipeline* se puede llevar a cabo, además de la paralelización de datos, una paralelización a nivel de tarea. Esto se consigue introduciendo paralelismo de hilos en las etapas del *pipeline,* o lo que es lo mismo, permitiendo que varios hilos ejecuten las distintas etapas.

La ejecución por varios hilos de cada etapa supone que el orden de salida del *pipeline* se altera respecto del orden de entrada, esto no es un problema en este caso ya que cada dato es independiente del resto.

La implementación de un *pipeline* con etapas paralelas se realiza a través de la función `parallel_pipeline`, esta función, de la biblioteca de TBB, recibe varios parámetros. El primero de ellos es un entero sin signo que indica el número de datos que pueden estar en el *pipeline* al mismo tiempo. Posteriormente recibe un número variable de argumentos de tipo `make_filter`, que definen cada una de las etapas.

La clase `make_filter` define los filtros o etapas que componen el *pipeline*. Está definida como una plantilla con dos parámetros que indican el tipo del objeto que recibe la etapa al comienzo de su ejecución y el tipo del objeto de salida al final de la misma, exceptuando la primera etapa, la cual no deberá recibir ningún objeto, ya que es la primera, y la última que no devolverá ningún objeto.. El constructor del objeto `make_filter` toma también dos parámetros, el primero de ellos es de tipo `filter::mode`, un enumerado compuesto de 4 valores:

- `parallel`: Indica que las operaciones de la etapa se pueden ejecutar por más de un hilo al mismo tiempo.
- `serial_out_of_order`: Indica que las operaciones de la etapa sólo pueden ejecutarse por un hilo al mismo tiempo.
- `serial_in_order`: Indica que las operaciones de la etapa sólo pueden ejecutarse por un hilo al mismo tiempo. Además, obliga a que en caso de que

los datos lleguen a la etapa en un orden distinto al orden de entrada su tratamiento y salida sea en el orden en que entraron al *pipeline*.

- `serial`: Se comporta exactamente igual que el anterior.

El segundo parámetro que toma el constructor es una función lambda que implementa la funcionalidad de la etapa. Como ya se ha explicado en el punto 2.8.5 de este documento, una función lambda es una función anónima semejante a la idea de *functor* o de puntero a función de C. Las funciones lambda declaradas en cada una de las etapas reciben y retornan valores del tipo indicado en la plantilla del objeto `make_filter`, exceptuando la primera que recibe como parámetro una referencia a un objeto de tipo `flow_control`. Este objeto es generado por la función `parallel_pipeline` y capturado por la función lambda del primer filtro que se declara. Este objeto sólo implementa una función, llamada `stop`, su funcionalidad es detener el *pipeline* cuando se ha terminado de insertar todos los datos.

En la siguiente figura se ejemplifica el uso de un *pipeline* para calcular el valor cuadrático medio.

```
float RootMeanSquare(float* first, float* last) {
    float sum=0;
    parallel_pipeline(16,
        make_filter<void,float*>(filter::serial,
            [&](flow_control& fc)-> float*{
                if( first<last ) {
                    return first++;
                } else {
                    fc.stop();
                    return NULL;
                }
            }
        ) &
        make_filter<float*,float>(filter::parallel,
            [](float* p){return (*p)*(*p);}
        ) &
        make_filter<float,void>(filter::serial,
            [&](float x) {sum+=x;}
        )
    );
    return sqrt(sum);
}
```

Figura 5.43. Ejemplo de *pipeline* simple.

En este caso el *pipeline* está formado por tres etapas y puede contener 16 elementos en ejecución. La función `RootMeanSquare` recibe dos punteros que apuntan al primer y

último valor de los elementos con los que se trabajará. Dentro de esta función se llama a `parallel_pipeline` para crear el *pipeline*, el cual está compuesto de 3 etapas. La primera etapa no recibe ningún parámetro, como ya se ha explicado, y devuelve un puntero a `float`. Es una etapa que se ejecutará de forma secuencial, es decir con un solo hilo y, además, la salida de los elementos se hará en el mismo orden en el que se han introducido, esto se indica con el primer parámetro que recibe el constructor, `filter::serial`.

En la función lambda que recibe como segundo parámetro se implementa la funcionalidad de esta etapa. La función lambda captura los elementos declarados en la función por referencia, por lo que dentro de ella podemos utilizar los punteros `first` y `last`, así como el valor `sum`. Además recibe por parámetro una referencia a un tipo `flow_control` que genera la propia función `parallel_pipeline`. Finalmente retorna un puntero a `float` como se había indicado en la plantilla de `make_filter`. Dentro de la etapa se va avanzando por la lista de valores comprendidos entre `first*` y `last*` y se va devolviendo un puntero a los mismos para que pasen a la siguiente etapa hasta llegar al último, en este punto se invoca la función `stop` del objeto tipo `flow_control` y se para la entrada de datos al *pipeline*, terminando este y la función `parallel_pipeline` cuando todas las etapas hayan acabado.

La segunda etapa se ejecuta de forma paralela, como indica su primer parámetro. En esta etapa no es necesaria sincronización entre hilos ya que cada dato es independiente de los demás y no es accedido por varios hilos de forma concurrente. En esta etapa no se necesita capturar ningún valor de la función principal, ya que sólo utiliza los punteros a cada uno de los valores sobre los que se aplica la operación y estos se reciben como parámetro de la etapa anterior. En la etapa se calcula el cuadrado de cada valor y se retorna a la siguiente etapa.

La última etapa se ejecuta de forma secuencial. Su función es hacer la suma de los cuadrados calculados en la etapa anterior. Para ello captura los valores de la función `RootMeanSquare` por referencia, esto le permite utilizar el valor `sum` para almacenar el resultado de la suma. La función recibe como parámetro un `float`, coincidente con el retorno de la etapa anterior.

Por último cuando el pipeline termina la función `RootMeanSquare` calcula la raíz cuadrada del resultado almacenado en `sum` y retorna este valor.

Para el proyecto la división en subtareas se ha llevado a cabo teniendo en cuenta las necesidades de los diferentes algoritmos de descarte. En la siguiente tabla se detalla la división propuesta para cada uno de los algoritmos.

	Etapa 1	Etapa 2	Etapa 3
Algoritmo I	Obtención de tarea. Creación de estructuras necesarias.	Ejecución de fase 1 del algoritmo. Preparación de fase 2 del algoritmo.	Ejecución de fase 2 del algoritmo. Finalización de tarea. Borrado de estructuras creadas en la subtarea 1.
Algoritmo II	Obtención de tarea. Creación de estructuras necesarias. Ejecución del descarte.	Ejecución de fase 1 del algoritmo. Preparación de fase 2 del algoritmo.	Ejecución de fase 2 del algoritmo. Finalización de tarea. Borrado de estructuras creadas en la subtarea 1.
Algoritmo III	Obtención de tarea. Creación de estructuras necesarias. Cálculo de los factores de descarte. Ejecución del descarte.	Ejecución de fase 1 del algoritmo. Preparación de fase 2 del algoritmo.	Ejecución de fase 2 del algoritmo. Finalización de tarea. Borrado de estructuras creadas en la subtarea 1.

Tabla 5.4. Subtareas realizadas en cada etapa del pipeline en los diferentes algoritmos.

Como se puede observar, el paso de parámetro entre etapas se hace a través de los retornos de las funciones que implementan, esto hace que haya que buscar formas alternativas cuando se quiere hacer pasar varios parámetros entre las etapas. En el caso que nos ocupa se hace necesario pasar de una etapa a otra 3 parámetros, el propio dato de que se va a tratar, el puntero a la cola donde pertenece el dato y un puntero al objeto `KFLib` que almacena el estado actual del objeto. La forma de conseguirlo es encapsular todos los datos en una clase o una estructura, en este caso, por simplicidad se ha decidido utilizar una estructura representada en la siguiente figura.

```
struct KFStruct{
    KFData <double> _data;
    KFLib<double>* _klib;
    KFQueue<double>* _queue;
    KFStruct(KFData <double> data, KFLib<double>* klib, KFQueue<double>*
    queue): _data(data), _klib(klib), _queue(queue){
        if(_klib == nullptr){
            cout << "Error en KFLib" << endl;
        }
    }

    ~KFStruct(){}
};
```

Figura 5.44. Estructura para encapsulación de datos

La implementación de esta técnica se ha llevado a cabo en dos funciones. La primera de ellas, llamada `pipeLine` se encarga de inicializar lo necesario, es muy similar a las funciones de inicio que se han utilizado en las técnicas anteriormente descritas. La función recibe como parámetros el número de hilos máximo que se quiere ejecutar y la ruta del fichero de datos. Al comienzo de la misma inicializa el planificador de TBB con el número de hilos que se le pasan por parámetro, salvo que este parámetro sea 0, en cuyo caso se inicializará con su constructor por defecto y será TBB quien decida cuál es el número idóneo de hilos.

Posteriormente ajusta los parámetros de ejecución de los hilos que se crearán en el *pipeline*, instancia las estructuras necesarias y lanza la ejecución de las funciones auxiliares que se encargan de insertar los datos y del descarte o parte de él en los algoritmos I y III.

Finalmente se invoca la función `parallelKalman`. En esta función es donde se ejecuta el *pipeline*. La función recibe un parámetro, una referencia al objeto `KFTaskManager` que contiene las colas de datos. Como se ha visto en la Tabla 5.4 las operaciones que se realizan en cada etapa dependen del algoritmo de descarte utilizado. A continuación se detallan estas operaciones en cada uno de ellos.

5.9.4.1 ALGORITMO I

En este caso en el pipeline solo se lleva a cabo el tratamiento de los datos, no se realiza ninguna tarea de descarte.

- Etapa 1: Esta etapa no recibe ningún parámetro y retorna un puntero al tipo `KFStruct`. La función lambda que implementa recibe como parámetro un tipo `flow_control` y captura por referencia los parámetros que recibe la función `parallelKalman`. La etapa ejecuta un bucle que realiza las siguientes operaciones.
    - Se obtiene un puntero de la cola circular. Se comprueba que la cola tenga datos, sino es así se devuelve el puntero y se extrae otro. Si la cola tiene datos se extrae el primero.
    - Se crea, en memoria dinámica, un objeto de tipo `KFStruct` con los datos que se han detallado anteriormente y se devuelve este objeto. Debe crearse en memoria dinámica ya que si no se destruiría al terminar La iteración del bucle.

- Etapa 2: Esta etapa recibe como parámetro el retorno de la anterior, es decir, un puntero a un tipo `KFStruct` y retorna este mismo valor. La función lambda que implemente toma como parámetro el puntero que recibe la etapa y además captura por referencia los parámetros que recibe la función principal. Sus operaciones son las siguientes.
    - Ejecuta la fase de predicción del algoritmo.
    - Prepara la ejecución de la fase de corrección ajustando los valores de las matrices de covarianzas.
- Etapa 3: Esta etapa recibe como parámetro el retorno de la anterior y no devuelva nada por ser la última. La función lambda toma como parámetro el que recibe la etapa y captura por referencia el parámetro de la función principal. Sus operaciones son las siguientes.
    - Ejecuta la etapa de corrección del algoritmo.
    - Reintroduce el puntero a la cola de datos en la cola circular.
    - Libera la memoria dinámica reservada en la primera etapa del *pipeline.*

Todas las etapas del *pipeline* son paralelas ya que el orden necesario en el tratamiento de los datos y los problemas de concurrencia ya han sido solucionados con el sistema de colas diseñado.

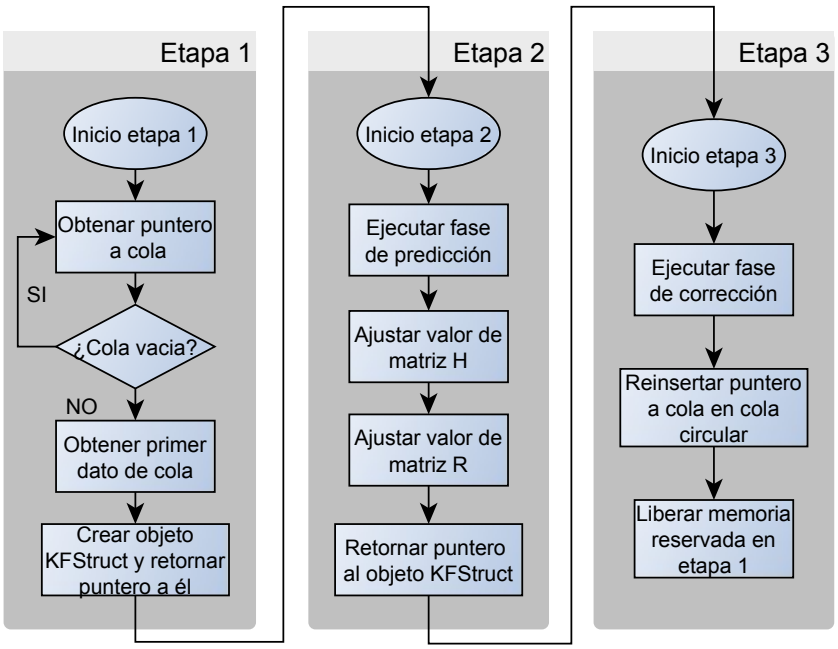

Figura 5.45. Diagrama de las etapas de *pipeline* para el algoritmo I

5.9.4.2 ALGORITMO II

En este caso en el *pipeline,* además de llevar a cabo el tratamiento de datos se lleva a cabo el descarte de los mismos.

- Etapa 1: Esta etapa no recibe ningún parámetro y retorna un puntero al tipo `KFStruct`. La función lambda que implementa recibe como parámetro un tipo `flow_control` y captura por referencia los parámetros que recibe la función `parallelKalman`. La etapa ejecuta un bucle que realiza las siguientes operaciones.
    - Se obtiene un puntero de la cola circular. Se comprueba que la cola tenga datos, sino es así se devuelve el puntero y se extrae otro. Si la cola tiene datos se extrae el primero.
    - Se toma el tiempo del sistema en ese instante. A ese tiempo tomado se le resta el tiempo en que se empezó a ejecutar, es decir, se calcula cuanto tiempo lleva el sistema en ejecución.
    - Se compara el tiempo de ejecución del sistema con la marca de tiempo del objeto que se va a para decidir el descarte que se debe aplicar.
    - Se aplica el descarte, descartando elementos como se indica en la Tabla 5.1.
    - Se crea, en memoria dinámica, un objeto de tipo `KFStruct` con los datos que se han detallado anteriormente y se devuelve este objeto.
- Etapa 2: Esta etapa recibe como parámetro el retorno de la anterior, es decir, un puntero a un tipo `KFStruct` y retorna este mismo valor. La función lambda que implemente toma como parámetro el puntero que recibe la etapa y además captura por referencia los parámetros que recibe la función principal. Sus operaciones son las siguientes.
    - Ejecuta la fase de predicción del algoritmo.
    - Prepara la ejecución de la fase de corrección ajustando los valores de las matrices de covarianzas.
- Etapa 3: Esta etapa recibe como parámetro el retorno de la anterior y no devuelva nada por ser la última. La función lambda toma como parámetro el

que recibe la etapa y captura por referencia el parámetro de la función principal. Sus operaciones son las siguientes.

- Ejecuta la etapa de corrección del algoritmo.
- Reintroduce el puntero a la cola de datos en la cola circular.
- Libera la memoria dinámica reservada en la primera etapa del *pipeline.*

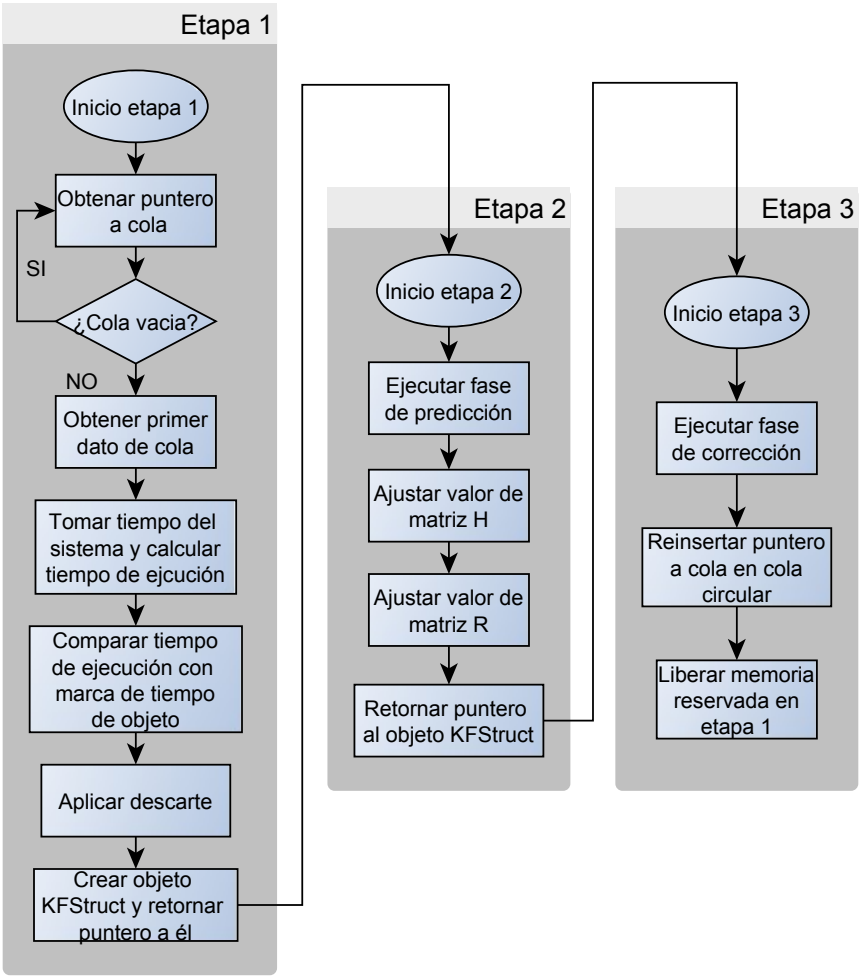

Figura 5.46. Diagrama de las etapas de *pipeline* para el algoritmo II

Al igual que en el algoritmo anterior, todas las etapas del *pipeline* son ejecutadas por varios hilos, consiguiendo paralelismo de tareas y de datos.

5.9.4.3 ALGORITMO III

Para el algoritmo III, el *pipeline* realiza tanto el tratamiento de datos como partes del descarte, en concreto calcula el factor de descarte y elimina los elementos descartados de las colas.

- Etapa 1: Esta etapa no recibe ningún parámetro y retorna un puntero al tipo `KFStruct`. La función lambda que implementa recibe como parámetro un tipo `flow_control` y captura por referencia los parámetros que recibe la función `parallelKalman`. La etapa ejecuta un bucle que realiza las siguientes operaciones.
    - Se obtiene un puntero de la cola circular. Se comprueba que la cola tenga datos, sino es así se devuelve el puntero y se extrae otro. Si la cola tiene datos se extrae el primero.
    - Se comprueba si el valor del factor de descarte para la cola es -1. Si es así se recalcula el factor de descarte para esa cola.
    - Se aplica el descarte, descartando tantos elementos como indica el factor de descarte de la cola.
    - Se crea, en memoria dinámica, un objeto de tipo `KFStruct` con los datos que se han detallado anteriormente y se devuelve este objeto.
- Etapa 2: Esta etapa recibe como parámetro el retorno de la anterior, es decir, un puntero a un tipo `KFStruct` y retorna este mismo valor. La función lambda que implemente toma como parámetro el puntero que recibe la etapa y además captura por referencia los parámetros que recibe la función principal. Sus operaciones son las siguientes.
    - Ejecuta la fase de predicción del algoritmo.
    - Prepara la ejecución de la fase de corrección ajustando los valores de las matrices de covarianzas.
- Etapa 3: Esta etapa recibe como parámetro el retorno de la anterior y no devuelva nada por ser la última. La función lambda toma como parámetro el que recibe la etapa y captura por referencia el parámetro de la función principal. Sus operaciones son las siguientes.
    - Ejecuta la etapa de corrección del algoritmo.
    - Reintroduce el puntero a la cola de datos en la cola circular.

- Libera la memoria dinámica reservada en la primera etapa del *pipeline.*

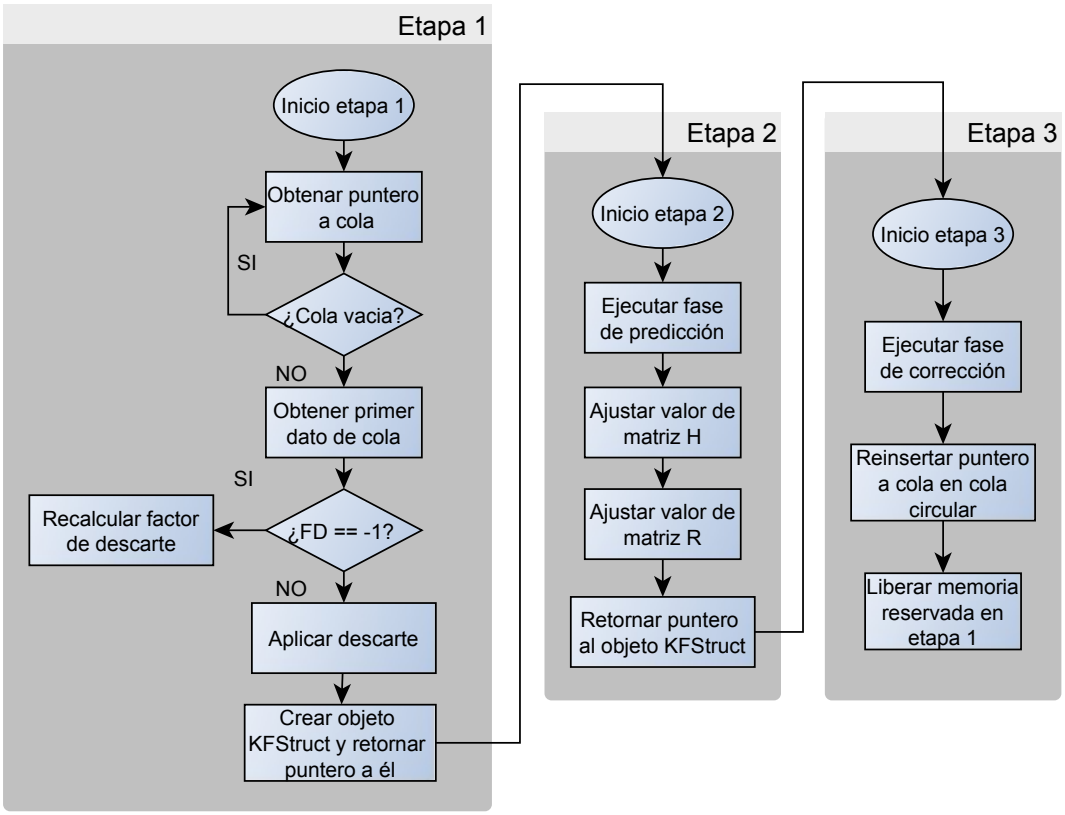

Figura 5.47. Diagrama de las etapas de *pipeline* para el algoritmo III

6 EVALUACIÓN

En este capítulo se muestran las pruebas que han sido llevadas a cabo y se analizan los resultados obtenidos. Las pruebas que se han realizado miden diferentes aspectos que se consideran importantes, como el rendimiento de la aplicación en diferentes máquinas y con distintas cargas de trabajo y la precisión que se logra respecto de la cantidad de datos que se descartan en cada algoritmo.

6.1 PRUEBAS DE RENDIMIENTO

Una de las pruebas que se han llevado a cabo ha sido medir el rendimiento de las diferentes implementaciones paralelas respecto de la versión secuencial. Para la realización de las pruebas se han utilizado varias máquinas con diferentes prestaciones.

Todas las máquinas en las que se ha probado el sistema son máquinas con arquitecturas de memoria compartida, siendo arquitecturas tanto CC-NUMA (*Cache Coherence - Non Uniform Memory Access*) como UMA (*Unifom Memory Access*). Las máquinas que se han utilizado son un Intel Core I5 con 4 núcleos in *hyperthreading* y 4 GB de memoria. Un Intel Core I7 con 4 cores con *hyperthreading* capaz de ejecutar 8 hilos simultáneos y 8 GB de memoria, y una máquina CC-NUMA compuesta por 4 sockets con procesadores Intel Xeon 7 de 6 núcleos con *hyperthreading* cada uno, lo que hace un total de 24 núcleos capaces de ejecutar 48 hilos simultáneos, esta máquina tiene además, 128 GB de memoria repartida en bancos de 32 GB, uno por *socket*.

Las pruebas que han sido realizadas se han basado en ejecutar las diferentes técnicas de paralelización con diferentes cargas de trabajo y diferente número de hilos para comprobar el número de elementos que es capaz de tratar el sistema y cuántos tiene que descartar así como la escalabilidad y la eficiencia de la aplicación.

Los datos que se han utilizado para ejecutar estas pruebas varían en cuanto a su cantidad, desde 500.000 muestras por segundo hasta 1.000.000 de muestras por segundo. En todos ellos se tiene una muestra cada milisegundo de cada objeto y varía el número de estos últimos, desde 600 hasta 1000 en saltos de 200 objetos.

6.1.1 PRUEBAS EN INTEL CORE I7

La primera máquina que se ha utilizado es una máquina con un procesador Intel Core I7 con 4 núcleos con *hyperthreading* a 3.30GHz, 8 GB de memoria y con un sistema operativo Ubuntu 12.04 LTS. Esta máquina tiene una arquitectura SandyBridge, las especificaciones técnicas completas se detallan en el Anexo I.

Las pruebas de rendimiento se han realizado con 3 niveles de carga como se indica en la Tabla 6.1.

Nivel de carga	Nº de datos
Bajo	6.000.000
Medio	8.000.000
Alto	10.000.000

Tabla 6.1. Niveles de carga utilizados en las pruebas.

6.1.1.1 PRUEBAS DEL ALGORITMO I EN VERSIÓN SECUENCIAL EN CORE I7

La Figura 6.1 muestra el número de datos tratados y descartados para una ejecución secuencial del algoritmo I con una carga de datos baja.

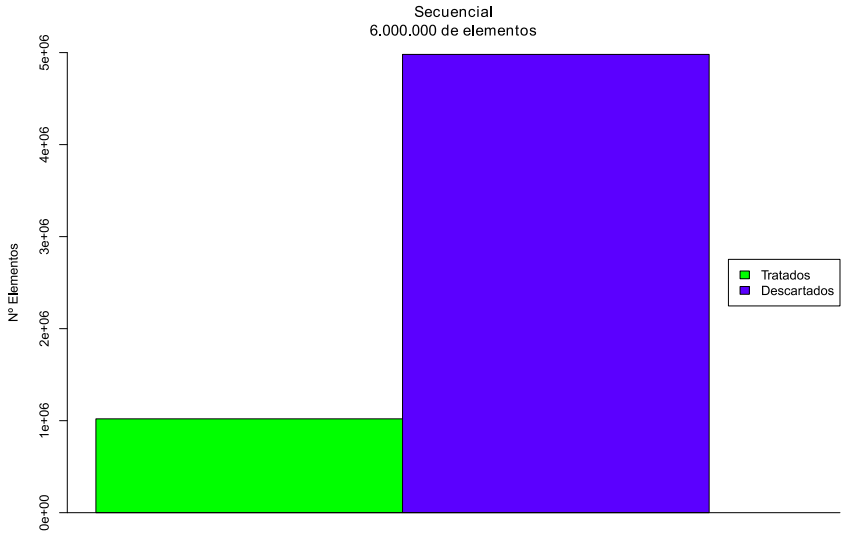

Figura 6.1. Resultado de la ejecución secuencial del algoritmo I con carga baja en Core I7.

Como se puede observar el número de datos descartados en la prueba es muy alto respecto del número de datos tratados, alrededor de un 85%.

En la Figura 6.2 se muestra el número de datos tratados y descartados para una carga media de datos.

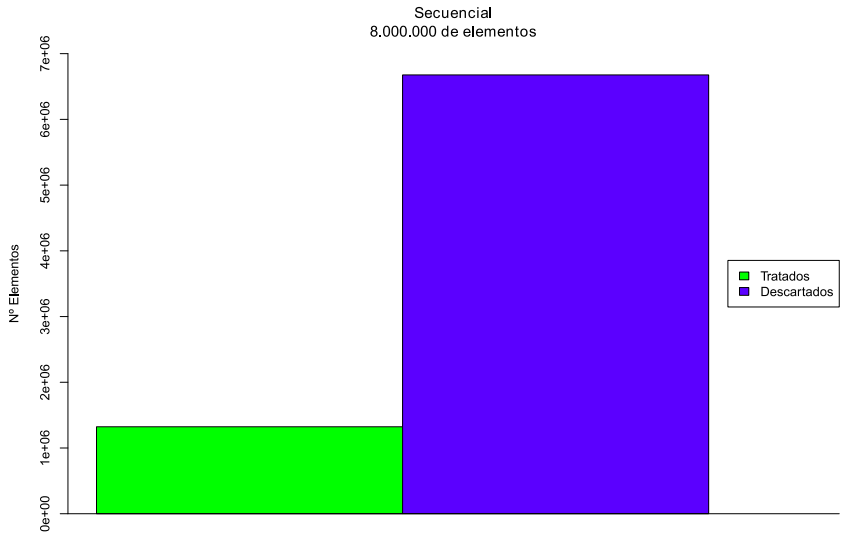

Figura 6.2. Resultado de la ejecución secuencial del algoritmo I con carga media en Core I7.

Para este nivel, como en el caso anterior, el número de datos descartados llega a valores en torno al 85% de descarte. Esto se traduce en que se ha tratado una cantidad mayor de datos, lo que se explica por la propia naturaleza del algoritmo de descarte, ya que cuantas más colas de datos halla más tardará el descarte en recorrerlas y vaciarlas y por lo tanto más tiempo tendrán los hilos de tratamiento para tratar datos. Sin embargo esto no es del todo bueno ya que puede suponer que se traten más datos retrasados.

La Figura 6.3 muestra el número de datos tratados y descartados para una carga alta de datos.

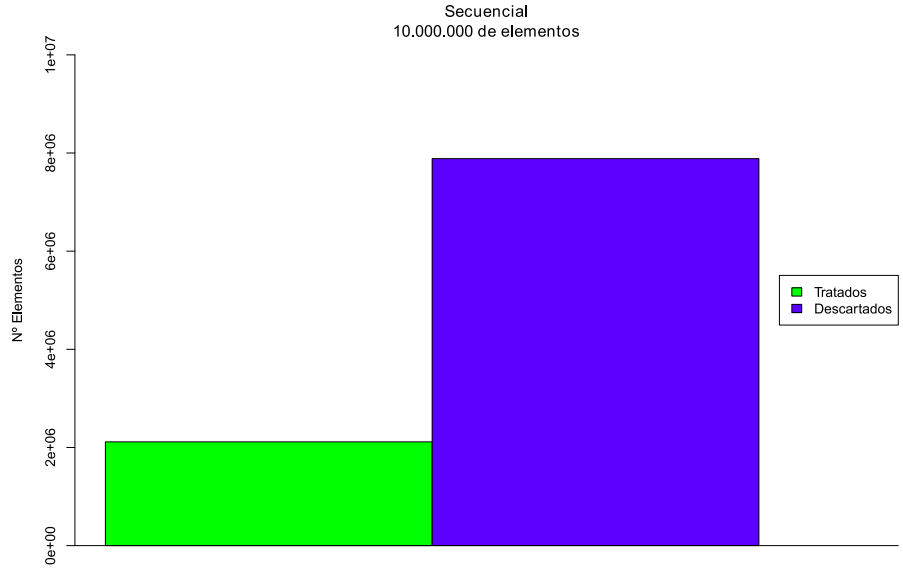

Figura 6.3. Resultado de la ejecución secuencial del algoritmo I con carga alta en Core I7.

En este nivel el porcentaje de datos descartados queda en torno al 80% La consecuencia de este comportamiento es la misma que en el caso anterior. No es que el sistema aumente su capacidad de tratamiento de datos, sino que disminuye su capacidad de descartar debido a que el número de colas con las que tiene que trabajar es muy alto.

Como se puede observar en las gráficas anteriores, la ejecución de forma secuencial del algoritmo produce que la mayoría de los datos de la simulación sean descartados porque no da tiempo a que sean tratados.

Con una media de datos perdidos cercana al 85% en los diferentes niveles que se han definido y probado, se puede considerar que el sistema no es capaz de trabajar con esas

cantidades de datos y que por tanto la implementación secuencial del mismo no es viable en ningún caso.

Como ya se ha comentado, las únicas formas que existen para acelerar de forma directa un algoritmo secuencial son la optimización del mismo y el incremento de frecuencia del procesador. La optimización del algoritmo secuencial ya se ha llevado cabo utilizando todas las herramientas que se han considerado necesarias. El incremento de la frecuencia de reloj del procesador es prácticamente imposible hoy en día, ya que se ha trabajado con uno de los procesadores con más frecuencia del mercado, 3,6 GHz, y como ya se ha explicado es muy difícil aumentar más la frecuencia debido al incremento de calor disipado que se produce.

La única solución para hacer este sistema viable es la paralelización del tratamiento de tareas, de forma que se aprovechen todos los núcleos y la potencia de cómputo de los procesadores actuales. En los siguientes puntos se muestran los resultados obtenidos para las versiones paralelizadas del algoritmo.

6.1.1.2 PRUEBAS DEL ALGORITMO I PARALELIZADO CON MODELO DE HILOS DE C++ EN CORE I7

A continuación se muestran los resultados obtenidos tras la paralelización con hilos nativos de C++ para una carga de datos baja.

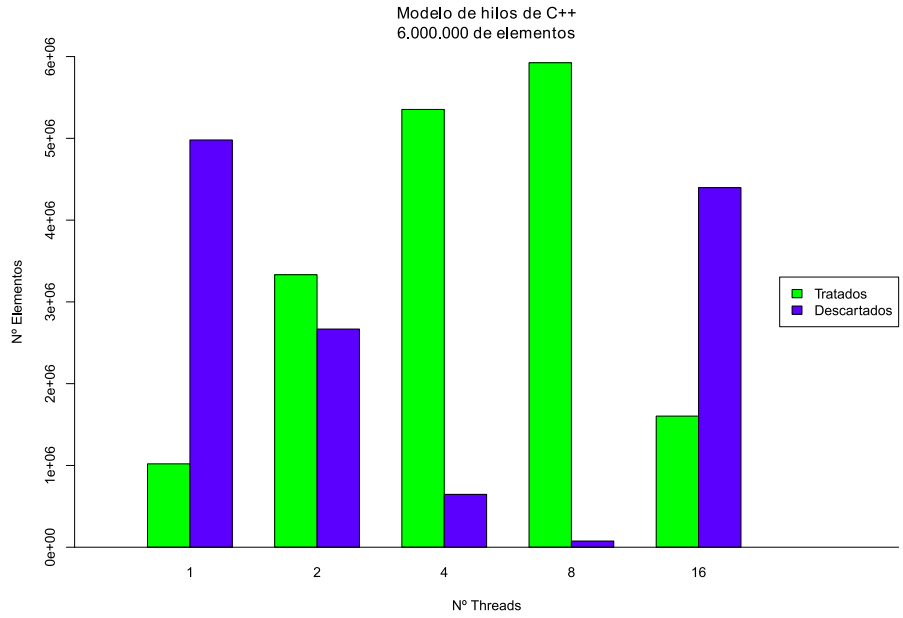

Figura 6.4. Resultado de la ejecución con hilos de C++ del algoritmo I con carga baja en Core I7.

Se observa que para la ejecución con 2 hilos se obtiene un número de elementos tratados algo más de 3 veces mayor que en el caso secuencial. Este fenómeno tiene que ver con la forma en que se realiza el descarte, ya que al aumentar la concurrencia aumenta el número de bloqueos y hace que estas funciones tarden algo más en ejecutarse y por tanto el tiempo que tienen los hilos para tratar datos es mayor.

El tamaño del problema no es lo suficientemente grande por lo que se observa una muy pequeña escalabilidad en el paso de 4 a 8 hilos.

Finalmente en el caso de ejecución con 16 hilos se observa en las dos pruebas que el rendimiento disminuye drásticamente. Esto es consecuencia de la sobrexplotación del sistema. Al ejecutar un número mayor de hilos que el número de procesadores se producen numerosos cambios de contexto con el consumo que éstos conllevan y por tanto el rendimiento disminuye. Una cuestión que acentúa este problema es que se está ejecutando en un procesador con *hyperthreading*. Esta tecnología es una forma de sobrexplotación del sistema

ya que en cada núcleo hay ciertos elementos que están duplicados, pero no todos, teniendo por lo tanto 16 hilos en menos de 8 núcleos.

En definitiva para este nivel de carga de datos se produce un incremento en el porcentaje de datos tratados de cerca del 85%, llegando a tratar prácticamente el 100% de los datos.

Para este nivel de datos la eficiencia en cuanto a datos tratados que se obtiene en el caso de 2 hilos es muy alta, ya que se tratan alrededor de 3 veces más datos por lo que se tiene una eficiencia cercana al 163%. Para el caso de 4 hilos la eficiencia también es muy alta ya que en ambas pruebas prácticamente se tratan todos los datos y se sitúa alrededor de un 130%. En el caso de 8 hilos la eficiencia disminuye, principalmente porque el tamaño del problema no es suficientemente grande y se sitúa en un 72%.

Aunque los resultados para este nivel de carga de datos no son muy representativos ya que el tamaño del problema no es suficientemente grande, se comienza a observar que la escalabilidad es buena pero no se va a mantener para un gran aumento de la concurrencia, ya que en cada paso que ésta se incrementa la eficiencia disminuye.

Para una carga media de datos se han obtenido los siguientes resultados.

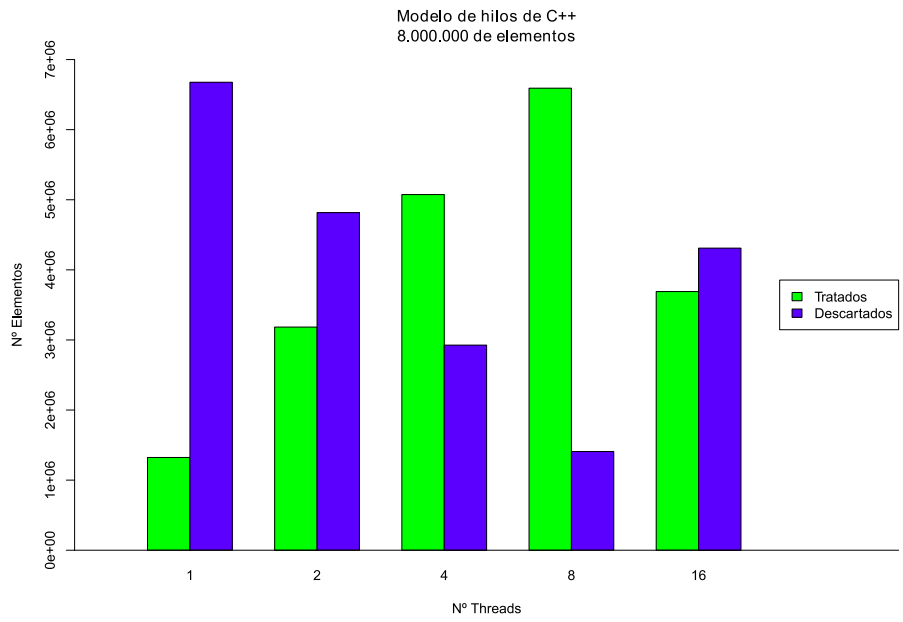

Figura 6.5. Resultado de la ejecución con hilos de C++ del algoritmo I con carga media en Core I7.

Para este caso de carga media, al aumentar el tamaño del problema lo suficiente se consigue dar trabajo a todos los hilos en el caso de ejecutar con 8. Esto hace que se produzca ganancia respecto a la ejecución con 4 hilos. También se puede deducir que para los casos de 2 y 4 hilos han llegado al tope de rendimiento con esta configuración de descarte, ya que el número de elementos tratados respecto al nivel de carga anterior no aumenta. En el caso de 8 hilos se observa que el número de elementos tratados aumenta ligeramente respecto al nivel anterior, por lo que se concluye que está bastante próximo a alcanzar su tope de rendimiento.

En cuanto al caso de ejecución de 16 hilos se vuelve a hacer patente que la sobrexplotación de los recursos del sistema produce una gran pérdida de rendimiento.

Para este nivel de carga de datos se ha obtenido un incremento en los datos tratados de un 70% ya que se tratan alrededor del 85% de los datos en el mejor de los casos.

Este resultado supone ser capaz de tratar 2,40 veces más datos que en el mismo nivel de carga para la implementación secuencial, teniendo por tanto una eficiencia del 120%.

En el caso de 4 hilos para este nivel se tratan 3,83 veces más datos que para el caso secuencial por lo que se tiene una eficiencia del 95%.

Para el caso de 8 hilos se tratan 4,98 veces más datos que en el caso secuencial, lo que supone una eficiencia del 62%.

Como se puede observar, utilizando esta técnica de paralelización el algoritmo es escalable, aunque probablemente no lo sea mucho más allá de 16 hilos sobre 16 núcleos ya que la eficiencia disminuye en cada incremento de la concurrencia.

En resumen, para este nivel de carga de trabajo se aprovechan mejor los recursos del sistema ya que se mantienen ocupados todos los procesadores disponibles. También se comprueba que para los casos de 2 y 4 hilos el máximo de datos que se pueden tratar son, en el primer caso alrededor de $3 \cdot 10^6$ elementos y en el segundo alrededor de $5 \cdot 10^6$ elementos.

En la Figura 6.6 se muestra el comportamiento del sistema con una carga de datos alta.

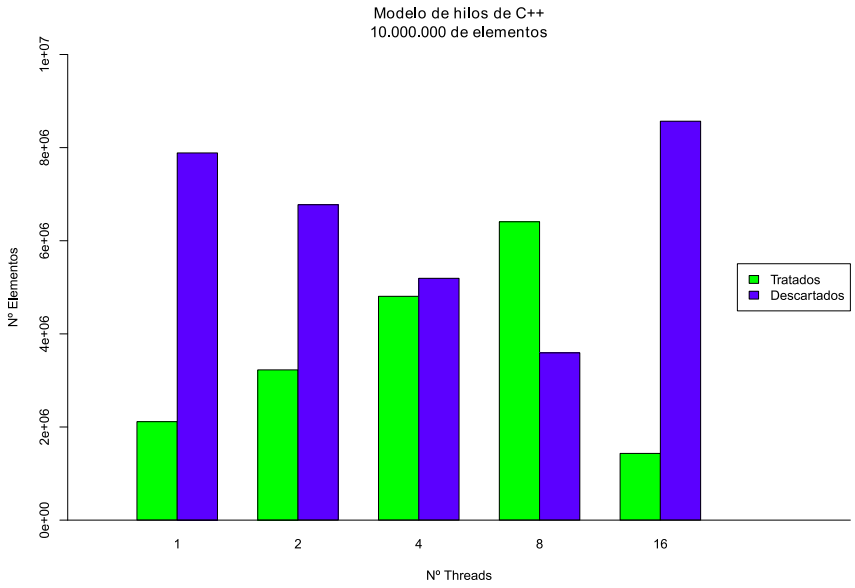

Figura 6.6. Resultado de la ejecución con hilos de C++ del algoritmo I con carga alta en Core I7.

Al igual que en los casos anteriores, en éste continúa la misma tendencia. Las ejecuciones con 2 y 4 hilos se mantienen en el máximo indicado anteriormente. Además, se observa que en el caso de la ejecución con 8 hilos el número de datos tratados es el mismo que para el caso anterior por lo que se deduce que también ha llegado a su máximo de rendimiento. El caso de 16 hilos muestra su comportamiento habitual.

Respecto a la ejecución secuencial se tiene un incremento del 44%, desde el 20% obtenido en la versión secuencial hasta el 64% obtenido en la versión paralela con 8 hilos.

Para este caso la eficiencia disminuye debido al incremento de datos tratados en la versión secuencial que se obtiene en esta prueba respecto a la primera. Para 2 hilos se tiene una eficiencia media del 76%. Para el caso de 4 hilos se tratan 2,27 veces más datos que en secuencial, lo que da una eficiencia del 57%. Para el último caso, con 8 hilos se tratan 3,03 veces más datos que en secuencial, obteniendo una eficiencia del 38%.

En resumen con esta técnica de paralelización se ha conseguido, en todos los niveles de carga de datos, un gran incremento en la capacidad de tratamiento de datos del sistema, llegándose a tratar prácticamente un 100% de los datos para niveles de carga baja, lo que supone un incremento para este nivel de carga del 80% respecto de la versión secuencial.

Para la prueba de carga media se ha logrado tratar en media un 85% de los datos en contraposición al 15% que se conseguía tratar en la versión secuencial para este nivel de carga, lo que supone un aumento del 70%.

Por último, para el nivel de carga alto se ha logrado una cantidad de datos tratados de alrededor del 64%, 44% más que la misma pruebas en secuencial en las que sólo se llego al 20%.

Como se puede observar la paralelización con esta técnica produce grandes ganancias en cuanto al número de datos tratados y, aunque a medida que la cantidad de datos aumenta las ganancias disminuyen, aún existe cierto margen.

Por otra parte, en cuanto a la eficiencia se observa que es muy buena para los casos probados, hasta 8 hilos, pero también se observa como disminuye paulatinamente con el aumento de la concurrencia, por lo que se puede estimar que para máquinas con más de 16 hilos la eficiencia será demasiado baja y por tanto no será escalable para esos niveles de concurrencia.

6.1.1.3 PRUEBAS DEL ALGORITMO I PARALELIZADO CON TBB EN CORE I7

En este punto se detallan los resultados obtenidos aplicando las diferentes técnicas de paralelización que ofrece la biblioteca Intel Threading Building Blocks. Al igual que en el caso anterior, las gráficas muestran el número medio de elementos tratados y descartados, los primeros se representan en verde y los segundos en azul.

Debido a que con TBB no es posible decidir el número de hilos, las pruebas sólo se pueden hacer con el número de hilos que la propia biblioteca decide como óptimo. Se ha comprobado que para esta máquina en concreto TBB considera óptimo el uso de 7 hilos de ejecución.

Se ha optado por mostrar en cada gráfica el resultado de todos los métodos de paralelización que se han usado, para de esta forma poder comparar en cada nivel de carga cual es más idóneo. Los niveles de carga que se han utilizado en estas pruebas son los indicados en la Tabla 6.1.

En la siguiente figura se muestran los resultados de la paralelización con TBB para una carga baja de datos. La gráfica muestra el resultado de ejecutar el sistema en forma secuencial, paralelizado con tareas de TBB, paralelizado con tareas encoladas de TBB y paralelizado con un *pipeline* de TBB.

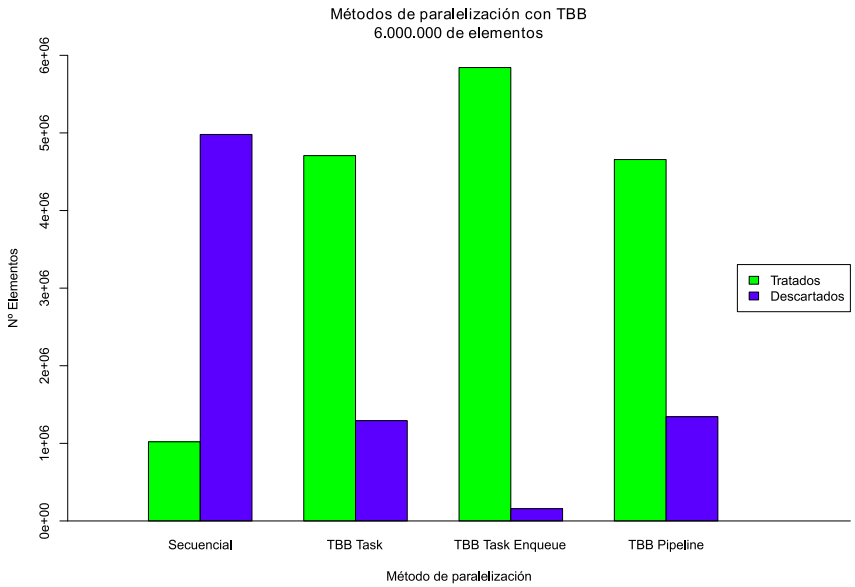

Figura 6.7. Resultado de la ejecución utilizando TBB del algoritmo I con carga baja en Core I7.

En la figura se observa como la opción *taskEnqueue* es capaz de tratar más datos que las otras dos, alrededor de un 20% más. Teniendo en cuenta que todas las opciones de paralelización ejecutan con el mismo número de hilos es posible que la implementación con *task* y la implementación con *pipeline* hayan llegado a su máximo de rendimiento.

Para este nivel de datos destaca la opción de *taskEnqueue* que ha sido capaz de tratar un 98% de los datos, lo que supone un 78% más que en la versión secuencial. Las otras dos opciones han llegado a tratar alrededor de un 75% de los datos.

La eficiencia de cada uno de los métodos es bastante alta. Para el primero y el tercero, se tiene que han tratado 4,6 veces más datos que en el modo secuencial, lo que produce una eficiencia del 66%. Para el *taskEnqueue* se tiene que ha tratado en la segunda prueba 5,7 veces más que en modo secuencial, es decir, un 82% de eficiencia.

La Figura 6.8 muestra los resultados para una carga de datos media. Al igual que en el caso anterior, en la implementación con *task* y *pipeline* el número de elementos tratados se mantiene. Además para la implementación basada en *taskEnqueue* se observa que llega a su tope ya que para esta cantidad de datos es capaz de tratar el mismo número que en el escalón anterior.

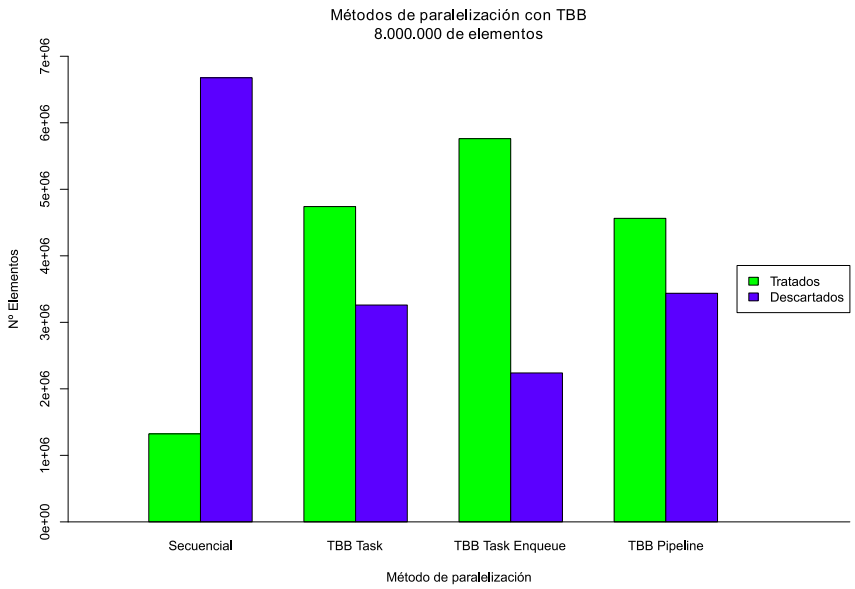

Figura 6.8. Resultado de la ejecución utilizando TBB del algoritmo I con carga media en Core I7.

Para este nivel el incremento de datos tratados respecto a la versión secuencial es de aproximadamente un 60% en los casos de *task* y *pipeline*. Para el caso de *taskEnqueue* este incremento se eleva hasta el 72% de datos tratados.

En cuanto a la eficiencia se tiene, para la implementación basada en *task,* una cantidad de datos tratados 3,58 veces mayor que en la implementación secuencial, obteniéndose una eficiencia del 51%. La implementación basada en *taskEnqueue* sobrepasa en 4,35 veces los datos tratados por la versión secuencial, obteniéndose con ella una eficiencia del 62%. Con la implementación basada en *pipeline* se tratan 3,44 veces más datos que en la versión secuencial, obteniendo en este caso una eficiencia del 49%.

Finalmente en el último nivel de carga todas las implementaciones han llegado a su tope y por lo tanto, como se puede observar, el número de datos tratados en cada una de ellas se mantiene constante respecto del nivel anterior. La Figura 6.9 muestra el resultado obtenido para este nivel de carga.

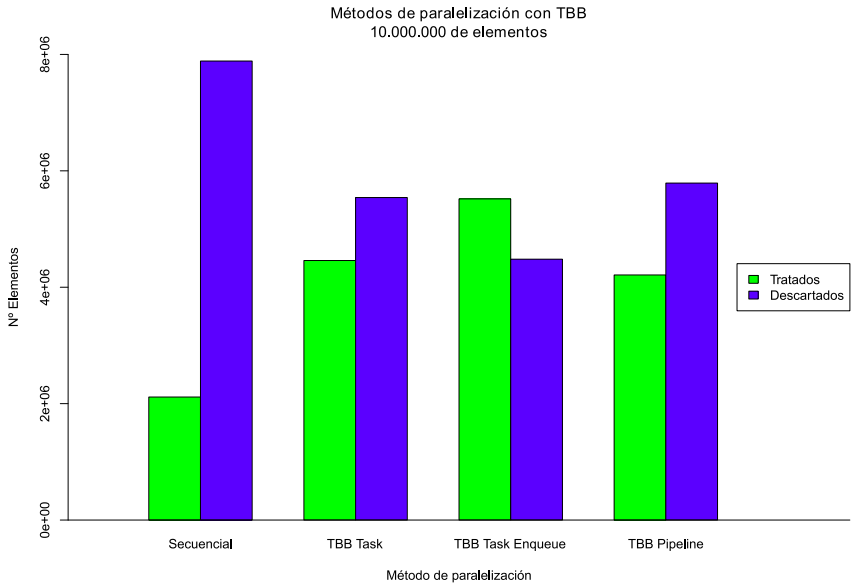

Figura 6.9. Resultado de la ejecución utilizando TBB del algoritmo I con carga alta en Core I7.

En este nivel de carga de datos tenemos que en la versión basada en *task* obtenemos un porcentaje de datos tratados del 45%, 25% más que en la versión secuencial. Para la versión con *taskEnqueue* un 55%, 35% más que en el caso secuencial. Por último con *pipeline* se obtiene un 42% de datos tratados, 22% más que en la versión secuencial.

En general TBB es una buena alternativa de paralelización debido a su facilidad de implementación y a que no es necesario que el programador gestione la creación y destrucción de hilos. Además, es muy portable, ya que sin tener que modificar el código se puede ejecutar en diferentes máquinas con distinta cantidad de procesadores y se adapta de forma automática al hardware subyacente.

Para todos los niveles de carga la opción con *taskEnqueue* es la que mejor comportamiento produce en términos de rendimiento, por lo que se tomará esta como referencia para comparar con el modelo de hilos de C++.

En cuanto al rendimiento, en general se encuentra por debajo de la implementación con hilos de C++, exceptuando la situación de una carga baja de datos, donde se tiene el mismo rendimiento. La Tabla 6.2 muestra el porcentaje de elementos tratados para la versión secuencial, la mejor versión con el modelo de hilos de C++ y la mejor versión con TBB.

Carga de datos	Secuencial	Hilos C++ (8)	TBB *TaskEnqueue*
6.000.000	17%	98%	98%
8.000.000	17%	85%	72%
10.000.000	21%	64%	55%

Tabla 6.2. Porcentaje de datos tratados en los mejores casos de cada prueba

Como se muestra en la tabla la implementación con 8 hilos de C++ mejora o al menos iguala el resultado ofrecido por la implementación *taskEnqueue* de TBB en todos los casos. La razón de este menor rendimiento por parte de TBB se encuentra en la ejecución de su planificador. El planificador de TBB está muy optimizado para trabajar en máquinas NUMA, sin embargo en máquinas UMA no genera ninguna ganancia y sí que necesita recursos para ejecutarse.

Otra de las razones que puede estar detrás de este menor rendimiento es que TBB considera para esta máquina que el número óptimo de hilos a utilizar es 7, mientras que las pruebas con hilos de C++ demuestran que existe escalabilidad hasta incluso más de 8 hilos.

Carga de datos	Hilos C++ (8)	TBB *TaskEnqueue*
6.000.000	72%	82%
8.000.000	62%	62%
10.000.000	38%	38%

Tabla 6.3. Eficiencia en los mejores casos de cada prueba

Esto último se refleja en la Tabla 6.3. Esta tabla muestra la eficiencia en cada prueba del modelo de hilos de C++ y del modelo de TBB con *taskEnqueue*. Como se puede observar la eficiencia es mayor o igual para el modelo de TBB en todos los casos, lo cual indica un mayor aprovechamiento de los recursos del sistema.

6.1.1.4 PRUEBAS DEL ALGORITMO II EN VERSIÓN SECUENCIAL EN CORE I7

A continuación se muestran los resultados obtenidos con la pruebas del algoritmo II. Los niveles de carga son los mismos que se definieron en la Tabla 6.1. Las gráficas muestran el número de elementos tratados, en una barra verde, y descartados, en una barra azul, para cada prueba.

La siguiente figura muestra el resultado de la ejecución secuencial para un nivel de carga bajo.

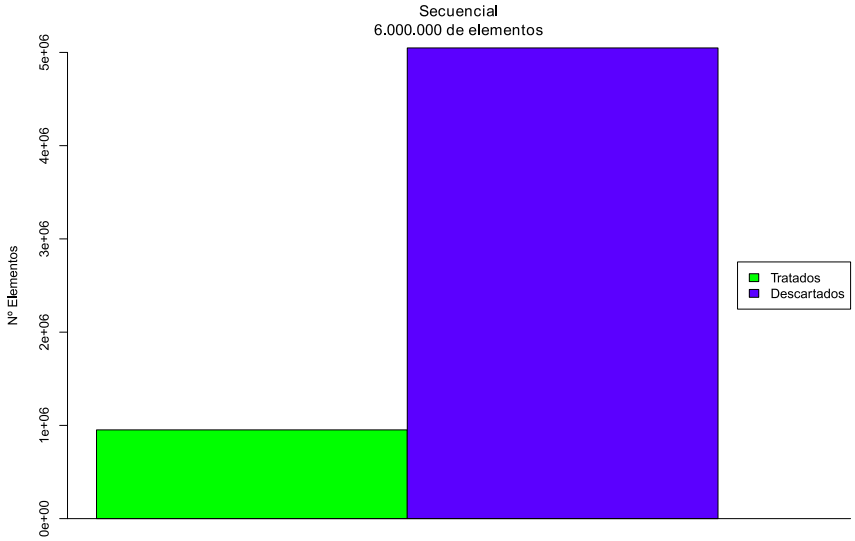

Figura 6.10. Resultado de la ejecución secuencial del algoritmo II con carga baja en Core I7

Al igual que en la prueba anterior la gráfica representada en la Figura 6.10 muestran como en modo secuencial sólo es posible tratar alrededor de un 16% de los datos. Un resultado muy similar al obtenido en el algoritmo I.

La Figura 6.11 muestra el resultado para una carga media de datos.

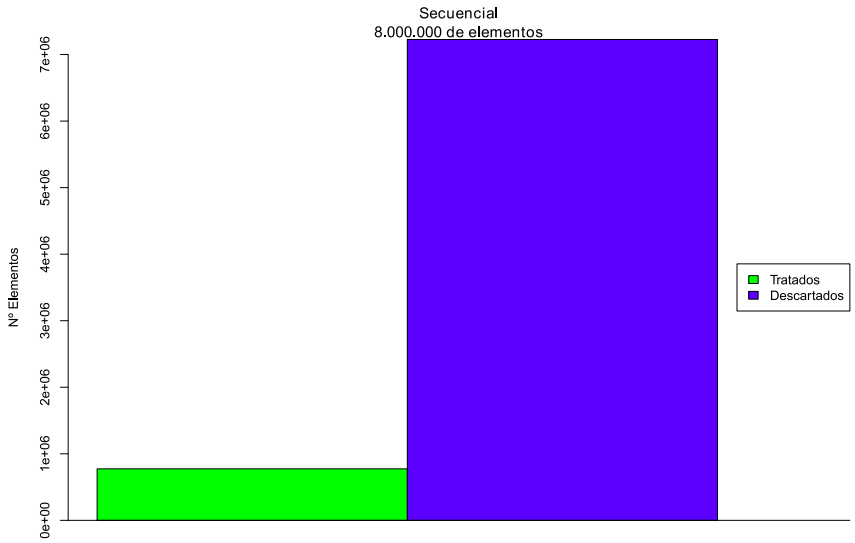

Figura 6.11. Resultado de la ejecución secuencial del algoritmo II con carga media en Core I7

Para este nivel el porcentaje de datos tratados es del 10%. Estos resultado son más bajos que en el caso del algoritmo I. Esto es debido a que para mantener el nivel de retraso bajo es necesario descartar más datos. Además se debe tener en cuenta que en este algoritmo, dado que implementa el descarte en los propios hilos de tratamiento, cuando se incrementa el descarte debido a que el retraso del sistema aumenta, se invierte más tiempo en realizarlo. Por lo tanto existe menos tiempo para el tratamiento de datos disminuyendo el número total de datos tratados al aumentar el nivel de carga.

La Figura 6.12 muestra el resultado para el último nivel de carga probado, carga alta.

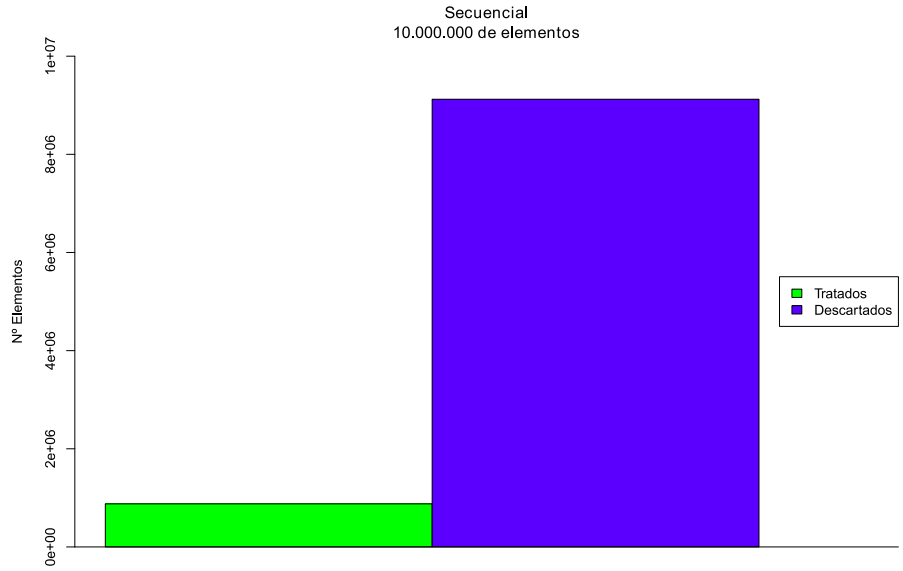

Figura 6.12. Resultado de la ejecución secuencial del algoritmo II con carga alta en Core I7.

Para este nivel de carga el porcentaje de datos tratados es de un 9%. Su comportamiento es igual que en el caso anterior, solo que en este nivel de carga no disminuye el número total de datos tratados, disminuye el porcentaje porque aumenta el número total de datos.

En general, al igual que en al caso del algoritmo anterior se tiene que el porcentaje de elementos tratados es muy bajo, con un máximo de 16% para carga baja de datos y del 9% para carga alta.

La siguiente tabla resume el porcentaje de datos tratados con la versión secuencial de cada algoritmo.

Carga de datos	Secuencial Algoritmo I	Secuencial Algoritmo II
6.000.000	17%	16%
8.000.000	17%	10%
10.000.000	21%	9%

Tabla 6.4. Elementos tratados en versión secuencial de algoritmos I y II en un I7.

Como se puede observar en la Tabla 6.4 la versión secuencial de este algoritmo siempre se sitúa por debajo de la misma versión del algoritmo I en cuanto a número de datos tratados. Sin embargo como ya se ha comentado en el punto 5.6.2 de este documento con

este algoritmo se consigue descartar datos de forma más homogénea y adaptándose más a la situación del sistema.

Como se puede ver cuanto más aumenta la carga del sistema menos datos se tratan, esto es debido ya que cuantos más datos existan más debe descartar el sistema para mantener el retraso en un nivel aceptable.

Cabe destacar que aunque este algoritmo descarte más datos que el anterior bajo las mismas circunstancias, los datos que se descartan en este caso se ajustan mejor a aquellos que el sistema no puede tratar en cada instante.

6.1.1.5 PRUEBAS DEL ALGORITMO II PARALELIZADO CON MODELO DE HILOS DE C++ EN CORE I7

A continuación se presentan los resultados obtenidos aplicando el modelo de hilos de C++.

Para una carga baja de datos se tienen los resultados mostrados en la siguiente figura.

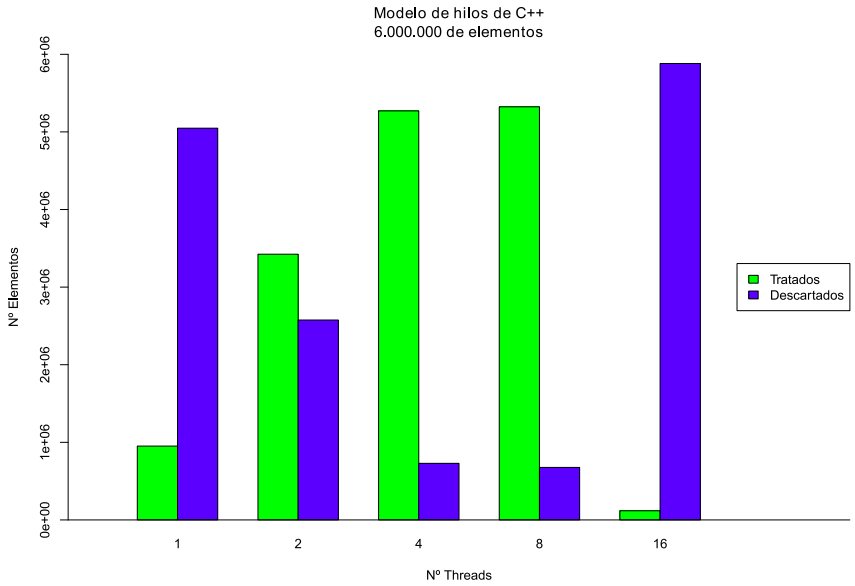

Figura 6.13. Resultado de la ejecución con hilos de C++ del algoritmo II con carga baja en Core I7.

Para este nivel de carga se observa que el algoritmo no es muy escalable a partir de 2 hilos ya que se tiene prácticamente el mismo resultado para la ejecución con 4 hilos que para la ejecución con 8 hilos, lo que indica que el tamaño del problema no es suficientemente grande para 8 hilos.

Al igual que anteriores pruebas el número de elementos tratados con 2 hilos es mucho mayor que el doble de los tratados en secuencial. Este comportamiento se produce porque al utilizarse intervalos para calcular el descarte puede ocurrir que un dato poco retrasado en una ejecución secuencial quede dentro del intervalo de descarte, mientras que en la ejecución multihilo este retraso se puede reducir aunque sea mínimamente y por tanto quedando este elemento fuera del descarte. Cuando se ejecuta con varios hilos, el hecho de que se consiga que ese mismo elemento no se retrase supone la oportunidad de tratar entre 25 y 50 elementos más, de los cuales es probable que se trate la mayoría.

El porcentaje de datos tratados con 2 hilos ha sido del 57%. Para el caso de 4 hilos se ha llegado al 88%. Por último en el caso de 8 hilos se ha obtenido un porcentaje de datos tratados del 89%. Para el caso de 16 hilos se observa que con este algoritmo la sobrexplotación se comporta peor que con el anterior y su porcentaje de datos queda por debajo incluso de la versión secuencial.

La eficiencia en cuanto a número de datos tratados en este nivel de datos se sitúa en un 178% con 3,6 veces más datos tratados para el caso de 2 hilos. Para la prueba con 4 hilos se han tratado 5,5 veces más datos que en el mismo nivel de la versión secuencial, obteniendo una eficiencia del 138%. En cuanto a la versión de hilos se tiene un incremento de datos tratados de 5,6 veces más que en modo secuencial, con una eficiencia del 70%.

Las Figura 6.14 muestra el resultado para una carga media de datos.

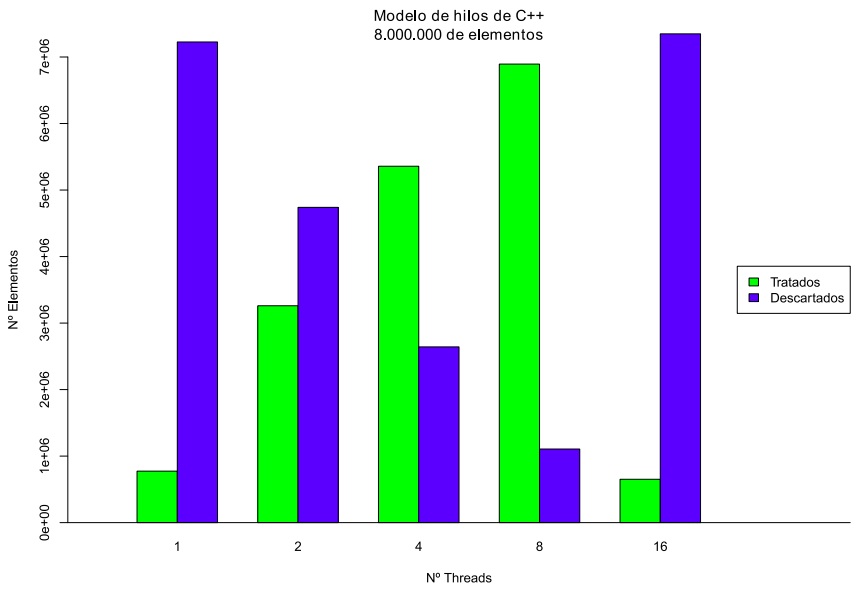

Figura 6.14. Resultado de la ejecución con hilos de C++ del algoritmo II con carga media en Core I7.

La figura muestra como para este nivel de carga de datos con 2 hilos se obtienen los mismos resultados que en el nivel anterior, es decir, se ha llegado al máximo rendimiento con esta cantidad de hilos. Los casos de 4 hilos muestran un comportamiento similar. La ejecución con 8 hilos supera en número de datos tratados a la de 4 hilos, esto es indicativo de que el tamaño del problema es acorde al número de hilos, al contrario que en el nivel anterior.

En este nivel el porcentaje de datos tratados por la ejecución con 2 hilos es del 40% del total. Para el caso de 4 hilos se obtiene un 67% de media de datos tratados. Por último con 8

hilos se obtiene un porcentaje de datos tratados del 86%. En la ejecución con 16 hilos se vuelve a poner de manifiesto que en este tipo de máquina la sobrexplotación del sistema produce una gran pérdida de rendimiento.

Como se puede observar el porcentaje de datos tratados se reduce respecto del nivel anterior. Esto se produce porque el algoritmo necesita descartar más datos para mantener el mismo nivel de retraso. Sin embargo hay que tener en cuenta que el número de datos tratados es mayor que en el nivel anterior, por lo que todavía se puede tener un cierto nivel de mejora.

La eficiencia en cuanto a datos tratados de este nivel de carga es, para el caso de 2 hilos es de un 200%. Para el caso de 4 hilos tenemos una eficiencia del 170%. Para el caso de 8 hilos la eficiencia es de un 112%.

La razón de que en este nivel la eficiencia sea tan alta es que la ejecución secuencial arrojó muy malos resultados.

En la Figura 6.15 se muestra el resultado obtenido para el último nivel de carga de datos.

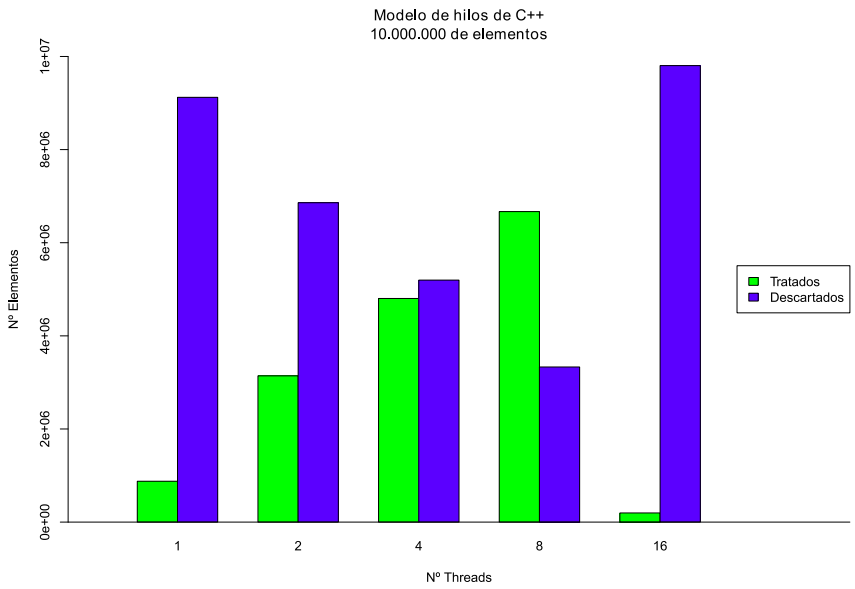

Figura 6.15. Resultado de la ejecución con hilos de C++ del algoritmo II con carga alta en Core I7.

Los resultados obtenidos con este último nivel de carga muestran que para todos los casos se ha llegado al tope máximo de datos que se pueden tratar con el modelo de hilos de C++.

Para este nivel de datos el porcentaje de datos tratados con 2 hilos es de un 31%. Para la ejecución con 4 hilos se obtiene un porcentaje de datos tratados del 48%. En el caso de 8 hilos el porcentaje llega al 66% del total. Para el caso de 16 hilos se repite el comportamiento de pruebas anteriores y se obtiene un rendimiento muy bajo.

Al igual que en caso anterior se observa una disminución en el porcentaje de datos tratados en las diferentes ejecuciones para mantener el nivel de retraso del sistema en un valor aceptable. Aunque a diferencia del nivel anterior no se incrementa el número de datos, por lo que el sistema ha llegado a su límite máximo de tratamiento de datos.

La eficiencia para este nivel de datos se sitúa en un 170% para el caso de 2 hilos, un 130% en el caso de 4 y un 95% en el caso de 8 hilos.

En general la paralelización de este algoritmo con el modelo de hilos de C++ ofrece muy buenos resultados en cuanto a rendimiento y eficiencia y permite tratar para todos los niveles de carga la mayoría de los datos.

Los niveles de eficiencia tan altos que se obtienen son debidos a que la versión secuencial del algoritmo descarta muchos datos para mantener el nivel de retraso bajo.

La Tabla 6.5 muestra el número de datos tratados utilizando 8 hilos de C++ para cada algoritmo. Se observa que para cargas de trabajo bajas el algoritmo I es capaza de tratar un mayor porcentaje de datos y para cargas altas ocurre lo contrario. Este comportamiento se debe a que para cargas bajas el algoritmo I, dado que no tiene en cuenta el retraso, es capaz de tratar todos los datos antes de que se active el hilo que realiza el descarte. En el caso del algoritmo II, que tiene en cuenta el retraso de cada dato que trata, sí que se produce descarte. Por otra parte en el algoritmo II al realizar el descarte en los propios hilos de tratamiento, estos toman más tiempo y por tanto pueden tratar menos datos.

En el caso de cargas altas, el algoritmo I además de los datos que descarta sin necesidad de ello, descarta los que no puede tratar, mientras que el algoritmo II ajusta los datos que descarta a los que no puede tratar descartando menos datos.

Carga de datos	Hilos C++(8) Algoritmo I	Hilos C++(8) Algoritmo II
6.000.000	98%	89%
8.000.000	85%	86%
10.000.000	64%	66%

Tabla 6.5. Comparativa de datos tratados por hilos de C++ en algoritmos I y II en I7.

6.1.1.6 PRUEBAS DEL ALGORITMO II PARALELIZADO CON TBB EN CORE I7

A continuación se presentan los resultados obtenidos aplicando las diferentes técnicas de paralelización de TBB al algoritmo II.

Para una carga baja de datos se obtienen los resultados de las siguientes 2 figuras.

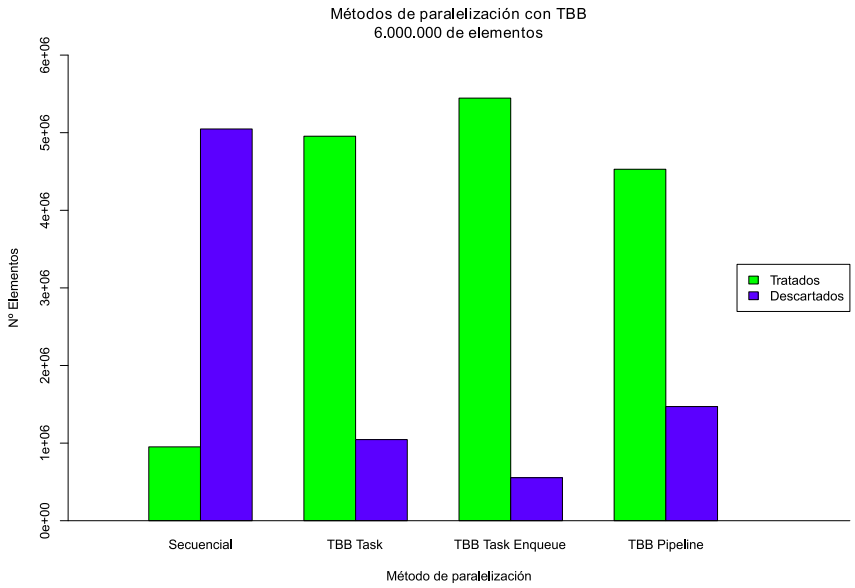

Figura 6.16. Resultado de la ejecución con TBB del algoritmo II con carga baja en Core I7.

Para este nivel de datos se observa que la implementación con *taskEnqueue* ofrece algo más de rendimiento que las otras dos opciones.

El porcentaje de datos tratados en este nivel es muy alto para todas las implementaciones. En el caso de la implementación con *task* el porcentaje de datos tratados llega al 83%. Para la implementación con *taskEnqueue* este porcentaje es del 91%. Por último la implementación *pipeline* es capaz de tratar un 75% de los datos.

La eficiencia en cuanto a los datos tratados para esta prueba se sitúa en el 74% en la implementación con *task* con un incremento de 5,2 veces en el número de datos tratados. Para la implementación con *taskEnqueue* se tienen 5,7 veces más datos tratados, con una eficiencia del 82%. Por último la eficiencia con la implementación basada en *pipeline* es del 68% con un incremento de 4,7 veces sobre la misma prueba en versión secuencial.

Para una carga media de datos se obtienen los resultados mostrados en la Figura 6.17.

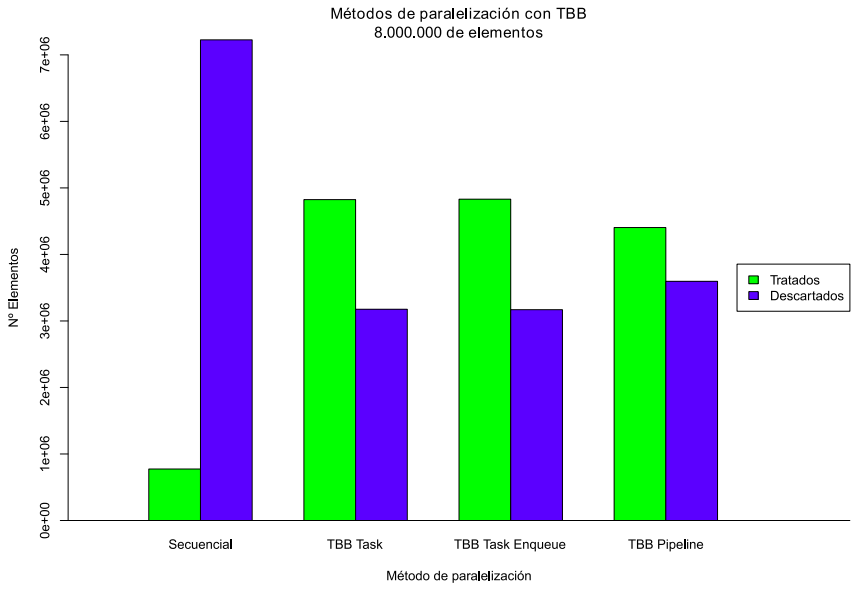

Figura 6.17. Resultado de la ejecución con TBB del algoritmo II con carga media en Core I7.

En la gráfica se aprecia como en la implementación con *taskEnqueue* se reduce el número de datos tratados, igualandose con la implementación basada en *task*.

En cuanto al porcentaje de datos tratados, en este nivel de carga se tiene un 60% en la implementación con *task.* Para la implementación basada en *taskEnqueue* este porcentaje es el mismo, el 60%. En la implementación con *pipeline* se obtiene un 55%. Estos porcentajes, más bajos que en el nivel anterior, y el hecho de que el número de datos tratados por cada implementación no haya aumentado indica que se ha llegado al tope máximo de la implementación.

La eficiencia en cuanto a datos tratados es del 89% para la implementación con *task* y *taskEnqueue*. Para la implementación con *pipeline* se obtiene un eficiencia del 81%.

Los resultados obtenido para un nivel alto de carga se muestran el la Figura 6.18.

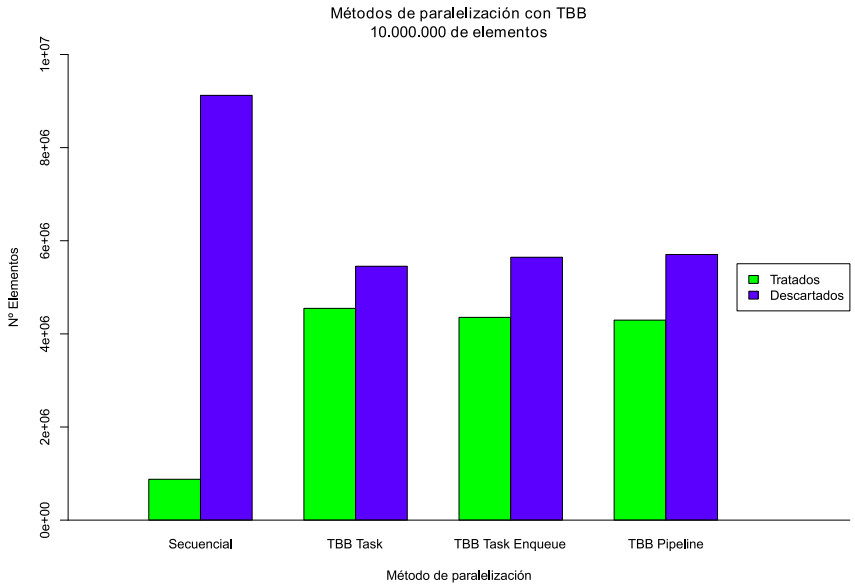

Figura 6.18. Resultado de la ejecución con TBB del algoritmo II con carga alta en Core I7.

En este nivel de carga todas las implementaciones han llegado a su máximo.

El porcentaje de elementos tratados en este nivel de carga disminuye respecto a los niveles anteriores, siendo en las implementaciones con *task* y *taskEnqueue* un 44% de los datos y un 43% en la implementación con *pipeline*.

La eficiencia para este nivel es de un 74% en el caso de *task*, un 70% para la implementación basada en *taskEnqueue* y un 69% para la implementación basada en *pipeline*.

En general la implementación con TBB produce buenos resultados en cuanto a rendimiento se refiere. Como muestra la Tabla 6.6 los resultados obtenidos por la implementación con TBB del algoritmo II son algo peor que los obtenidos por el algoritmo I. Este comportamiento se debe a que el algoritmo II implementa el descarte en los hilos de tratamiento y por tanto estos hilos tienen mayor carga.

Carga de datos	TBB (*taskEnqueue*) Algoritmo I	TBB (*taskEnqueue*) Algoritmo II
6.000.000	98%	91%
8.000.000	72%	60%
10.000.000	55%	44%

Tabla 6.6. Comparativa de datos tratados por TBB en algoritmos I y II.

En cuanto al rendimiento, en general se encuentra por debajo de la implementación con hilos de C++, exceptuando la situación de una carga baja de datos, donde se tiene un rendimiento algo superior en TBB. La Tabla 6.7 muestra el porcentaje de elementos tratados para la versión secuencial, la mejor versión con el modelo de hilos de C++ y la mejor versión con TBB.

Carga de datos	Secuencial	Hilos C++ (8)	TBB *TaskEnqueue*
6.000.000	16%	89%	91%
8.000.000	10%	86%	60%
10.000.000	9%	66%	44%

Tabla 6.7. Porcentaje de datos tratados en los mejores casos de cada prueba para algoritmo II en I7.

Como se muestra en la tabla la implementación con 8 hilos de C++ mejora el resultado ofrecido por la implementación *taskEnqueue* de TBB en la mayoría de los casos. En los que C++ está por debajo la diferencia es muy pequeña. Este resultado como ya se ha descrito anteriormente es en parte culpa del planificador de TBB. Por otra parte no se debe olvidar que TBB utiliza sólo 7 hilos.

Carga de datos	Hilos C++ (8)	TBB *TaskEnqueue*
6.000.000	70%	82%
8.000.000	112%	89%
10.000.000	95%	70%

Tabla 6.8. Eficiencia en los mejores casos de cada prueba para el algoritmo II en I7.

La Tabla 6.8 muestra la eficiencia en cada prueba del modelo de hilos de C++ y del modelo de TBB con *taskEnqueue*. Como se puede observar la eficiencia es mayor o igual para el modelo de C++ en casi todos los casos lo cual indica una mayor escalabilidad por parte de este modelo en cuanto a tamaño del problema se refiere.

6.1.1.7 PRUEBAS DEL ALGORITMO III EN CORE I7

Como se ha explicado en el punto 5.6.3 de este documento el algoritmo III se basa en llevar un histórico de los últimos 10 segundos de ejecución. A partir de este histórico el algoritmo puede calcular cuántos datos es capaz de tratar en un determinado intervalo de tiempo o iteración. Sabiendo este número de datos que se pueden tratar, el número de objetos móviles que existen en el sistema y el número de datos que se tiene de cada objeto en las colas, se puede calcular un valor, llamado factor de descarte, que indica el número de datos que se deben descartar por cada dato que se trate.

Como se puede observar este algoritmo descarta un número de datos muy ajustado al número de datos que se pueden tratar evitando que el sistema se retrase. Además, el algoritmo puede adaptarse a las diferentes circunstancias que se den en el sistema. Para el correcto funcionamiento de este algoritmo, es necesario que la simulación tenga una duración suficiente como para poder crear este histórico y que el algoritmo converja y aproveche al máximo el hardware. Sin embargo debido a las limitaciones del propio hardware no ha sido posible llevar a cabo una prueba lo suficientemente larga, en cuanto a duración, como para que el algoritmo converja.

El problema que se ha encontrado ha sido en la cantidad de memoria del sistema. Como ya se ha dicho, esta máquina tiene una memoria de 8GB, para llevar a cabo la prueba, se estima que la duración de la simulación debería ser algo mayor de 1 minuto. Cada dato, ocupa en memoria cerca de 180 bytes. Para realizar las pruebas de rendimiento que se han realizado se deberían utilizar dependiendo del nivel de carga, los datos de la Tabla 6.9.

Carga de datos	Datos por segundo	Datos en un minuto
Bajo	600.000	36.000.000
Medio	800.000	48.000.000
Alto	1.000.000	60.000.000

Tabla 6.9. Cantidad de datos a utilizar en los diferentes niveles del algoritmo III.

El espacio en memoria necesario para almacenar estos datos va desde algo más de 6 GB, para el caso de un nivel de carga bajo, hasta algo más de 10GB, en el caso de un nivel de carga alta de datos.

Además de este espacio debe considerarse el espacio necesario para almacenar las estructuras de datos y el espacio que utiliza el propio sistema operativo. Toda esta cantidad de

espacio hace que se necesite para ejecutar un minuto de simulación alrededor de 9,5GB en el caso de un nivel de carga bajo y cerca de 14GB para el nivel de carga alto.

Esta cantidad de memoria no está disponible en el sistema, por lo que se produce hiperpaginación. Esta hiperpaginación produce que el rendimiento del sistema caiga rápidamente y que en ocasiones el sistema se quede bloqueado. Debido a esta circunstancia sólo se ha podido llevar a cabo las pruebas para un nivel de carga de datos bajo, ya que en al ejecutar las pruebas de los otros 2 niveles el sistema se quedaba bloqueado. Los resultados de estas pruebas se exponen a continuación.

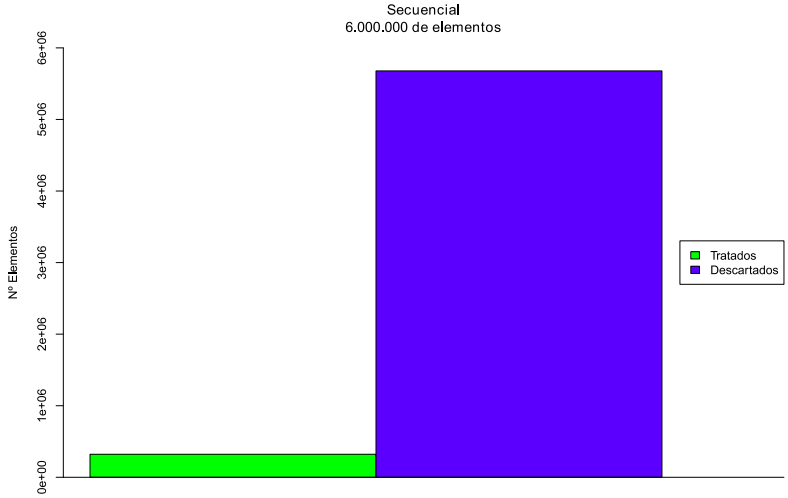

Figura 6.19. Resultado de la prueba del algoritmo III con carga baja en Core I7.

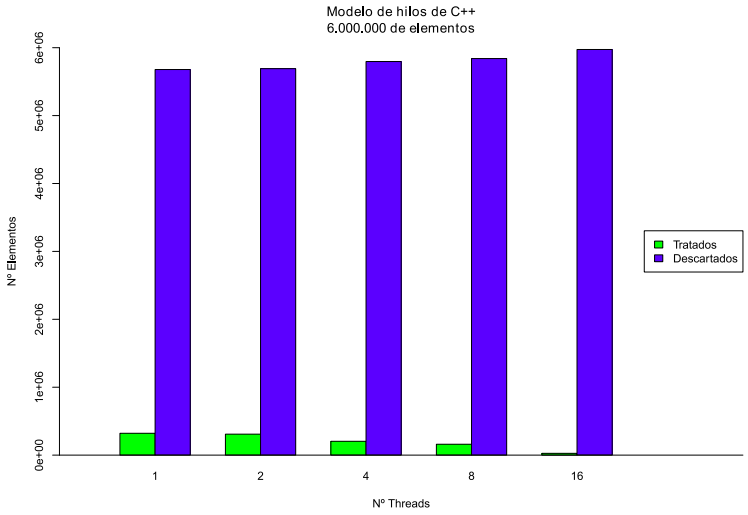

Figura 6.20. Resultado de la prueba del algoritmo III con carga media en Core I7.

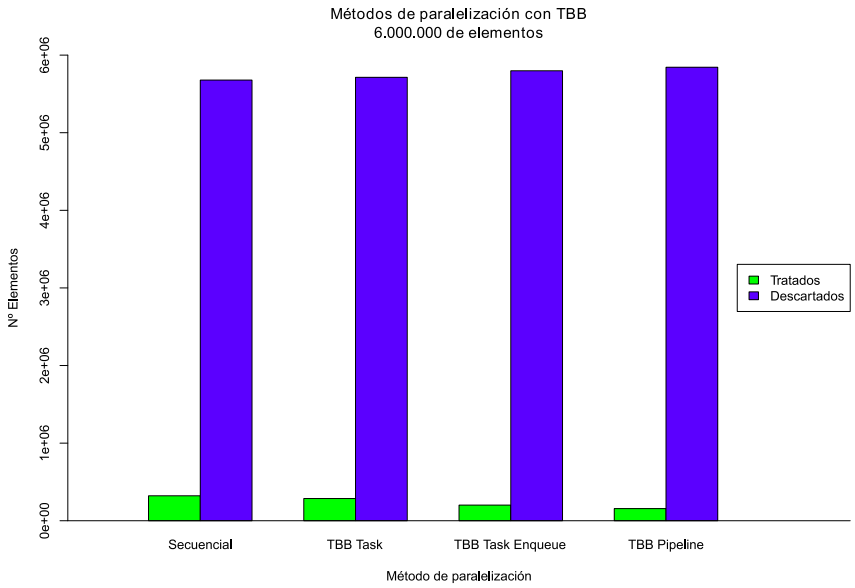

Figura 6.21. Resultado de la prueba del algoritmo III con carga alta en Core I7.

Como se observa en las 3 figuras anteriores el rendimiento del algoritmo III es bajísimo. Tratando cómo máximo el 5 ó 6% de los datos totales en el mejor de los casos. Este comportamiento está producido, como ya se ha dicho por la paginación que produce al no poder albergar todos los datos en memoria principal, debido a que no caben y tener que utilizar la memoria de intercambio del sistema, alojada en el disco. Esta paginación produce varios efectos:

- Al llevar a cabo la inserción, si los datos que se van a insertar no están en memoria principal, es necesario acceder al disco para obtenerlos. Este acceso lleva mucho tiempo por lo que el hilo encargado de insertar los datos se retrasa y por tanto estos no llegan a tiempo, pudiendo suceder, sobre todo en los primeros instantes de la ejecución, que los hilos que tratan los datos se encuentren las colas vacías y se desperdicie tiempo de procesamiento.
- Otra circunstancia que se puede dar, con bastante frecuencia, es que cuando un hilo de tratamiento toma un puntero a una cola de datos, los datos de esta no están en memoria principal, sino en memoria de intercambio. Al traerlos desde disco, además del tiempo que conlleva esta operación, es necesario conseguir espacio en memoria principal para estos datos. Esto lo consigue el S.O expulsando otros datos que en ese momento estén en memoria, esta operación también lleva tiempo, porque hay que copiar los datos a disco. Posteriormente cabe la

posibilidad de que otro hilo quiera acceder a esos datos recién expulsados, teniendo que acceder a disco de nuevo y repetir el proceso anterior. Este problema se agudizará cuantos más hilos de tratamiento existan ya que más operaciones de memoria se generarán.

6.1.2 PRUEBAS EN INTEL CORE I5

La primera máquina que se ha utilizado es una máquina con un procesador Intel Core I5 con 4 núcleos a 3.30GHz, 4 GB de memoria y con un sistema operativo Ubuntu 12.04 LTS. Esta máquina tiene una arquitectura SandyBridge, las especificaciones técnicas completas se detallan en el Anexo I.

Las pruebas de rendimiento se han realizado con 3 niveles de carga como se indica en la Tabla 6.1.

6.1.2.1 PRUEBAS DEL ALGORITMO I EN VERSIÓN SECUENCIAL EN CORE I5

A continuación se muestran los resultados obtenidos en la ejecución secuencial del sistema. En cada gráfica se representan mediante una barra verde el número medio de datos tratados por segundo y mediante una barra azul el de datos descartados.

La Figura 6.22 muestra los resultados de la ejecución con una carga de datos baja.

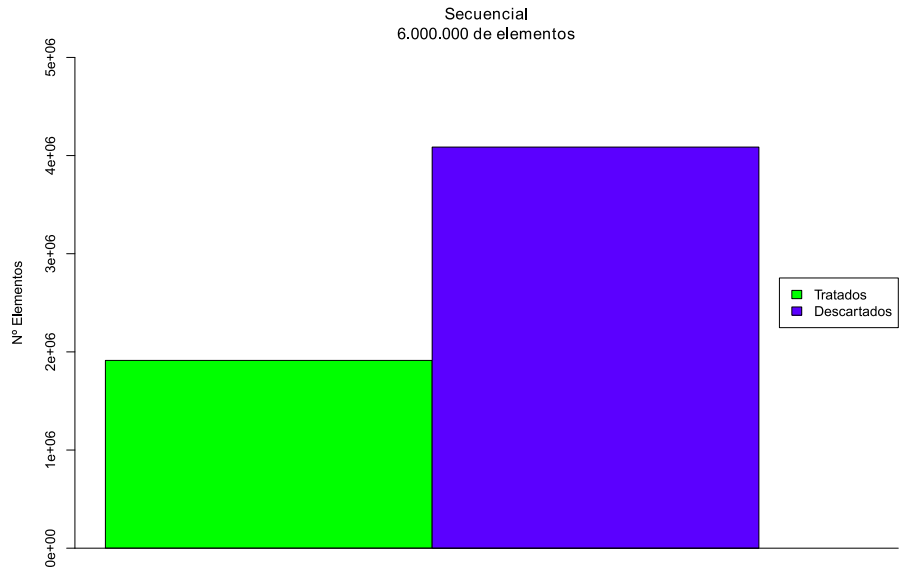

Figura 6.22. Resultado de la ejecución secuencial del algoritmo I con carga baja en Core I5.

En la gráfica se observa como el número de datos tratados es muy bajo respecto del total, produciéndose un descarte de alrededor del 75% de los datos.

En la siguiente figura se presenta los resultados de la ejecución para una carga media de datos. En este nivel el único cambio que se produce es el incremento de datos descartados, el cual se sitúa en un 80%.

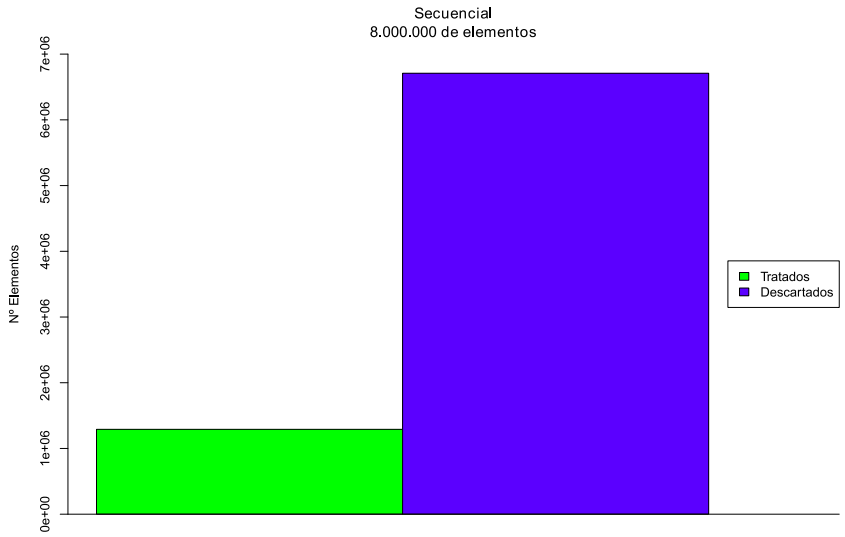

Figura 6.23. Resultado de la ejecución secuencial del algoritmo I con carga media en Core I5.

El último nivel de carga de datos, representado en la Figura 6.24 muestra la misma tendencia que se ha dado en los dos niveles anteriores, el número de datos que el sistema es capaz de tratar se mantiene y por tanto el porcentaje de datos descartados se incrementa llegando a situarse cerca del 85% del total.

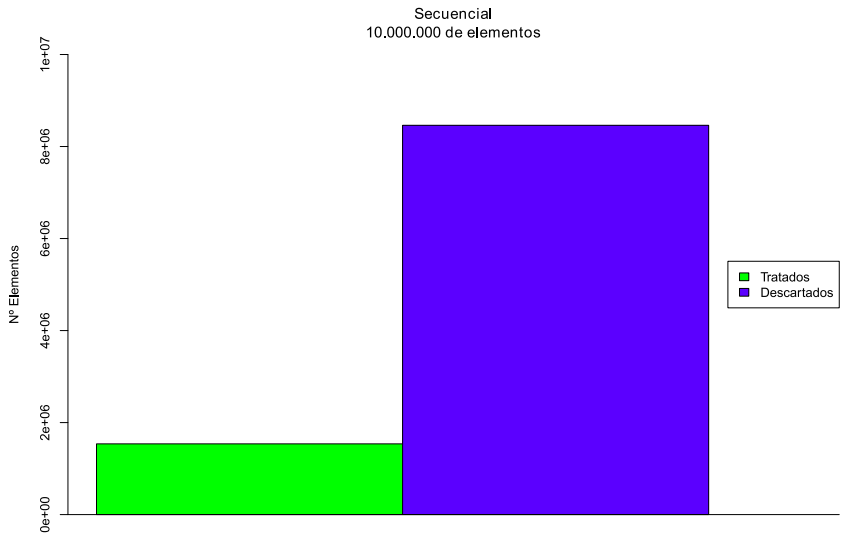

Figura 6.24. Resultado de la ejecución secuencial del algoritmo I con carga alta en Core I5.

En todos los casos se observa que porcentaje de datos tratados es muy pequeño respecto al total, teniendo como máximo alrededor del 25%. Esto implica que el sistema no sea viable debido a que se descartan la mayor parte de los datos. Como ya se ha explicado es imposible, a día de hoy, conseguir una ganancia suficiente de cómputo aumentando la frecuencia de reloj del procesador, por lo que para conseguir un sistema viable en esta máquina es necesario aprovechar su arquitectura tratando los datos en paralelo.

6.1.2.2 PRUEBAS DEL ALGORITMO I PARALELIZADO CON MODELO DE HILOS DE C++ EN CORE I5

En este punto se muestran los resultados obtenidos aplicando paralelización con el modelo de hilo de C++. Los gráficos muestra el número de datos tratados, en verde, y descartados, en azul.

A continuación se muestran los resultados obtenidos tras la paralelización con hilos nativos de C++ para carga de datos baja.

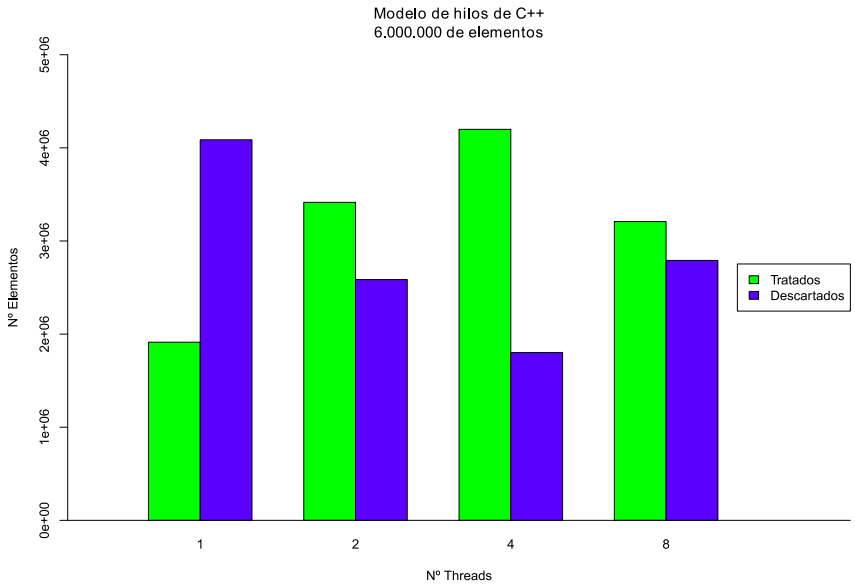

Figura 6.25. Resultado de la ejecución con hilos de C++ del algoritmo I con carga baja en Core I5.

En el gráfico se observa como en todos los casos el número de elementos tratados llega es muy superior al de la versión secuencial.

Para el caso de 2 hilos tenemos un porcentaje de elementos tratados del 57%. En el caso de 4 hilos se tiene un porcentaje del 70%. En el caso de 8 hilos el rendimiento baja a niveles de 2 hilos debido a la sobrexplotación del sistema obteniendo un 53% de datos tratados.

En cuanto a la eficiencia respecto del número de elementos tratados se obtiene, para este nivel de datos un 90% para la ejecución con 2 hilos, un 55% para la ejecución con 4 hilos y un 21% en la ejecución con 8 hilos.

Para este nivel se matienen buenos niveles de eficiencia exceptuando el caso de 8 hilos que como ya se ha visto en pruebas anteriores reduce mucho el rendimiento debido a que se están sobreutilizando los recursos del sistema. Sobre esta sobrexplotación cabe destacar que la dsiminución de rendimiento en esta máquina es menor que en la anterior debido a que esta máquina no tienen *hyperthreading* que como ya se ha visto es una forma de sobrexplotación parcial.

La siguiente figura muestran el resultado para una carga media de datos.

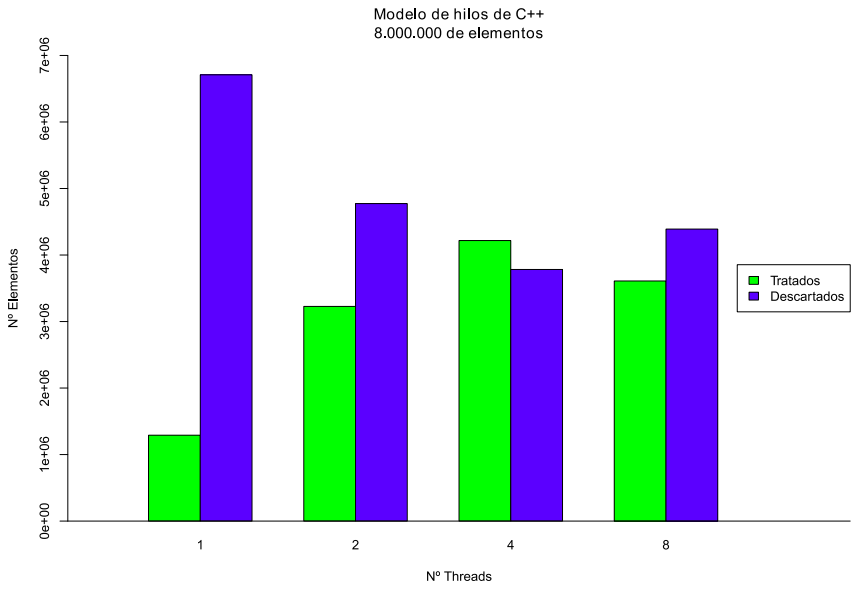

Figura 6.26. Resultado de la ejecución con hilos de C++ del algoritmo I con carga media en Core I5.

En este nivel de carga se mantiene el número de datos tratados en todos los casos y por tanto el número de descartados aumenta, disminuyendo así el porcentaje de datos tratados. Este porcentaje en el caso de 2 hilos es del 40%. Para el caso de 4 hilos se tiene un porcentaje de datos tratados del 53%. Por último en el caso de 8 hilos se tiene un porcentaje del 45%.

La eficiencia en cuanto a datos tratados para este nivel de datos es, en el caso de 2 hilos es del 125%. En el caso de 4 hilos se obtienen una eficiencia del 82%. Por último para 8 hilos se obtiene una eficiencia del 35%. Este incremento de eficiencias en todos los casos viene dado por el menor número de elementos tratados en la versión secuencial.

En la Figura 6.27 se muestra el resultado obtenido para una carga alta de datos.

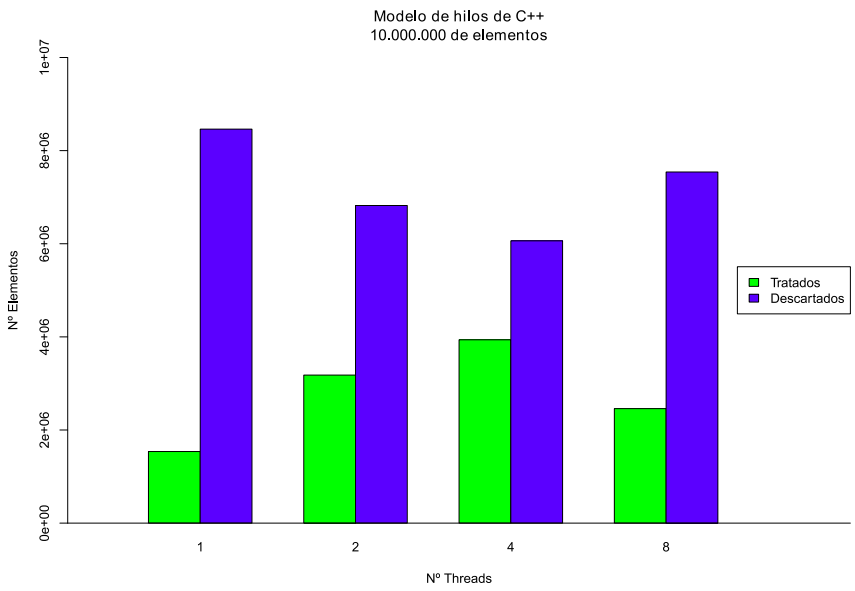

Figura 6.27. Resultado de la ejecución con hilos de C++ del algoritmo I con carga alta en Core I5.

En este nivel se obtiene el mismo comportamiento que en el anterior, aumenta el número de elementos descartados y el de tratados se mantiene.

El porcentaje de elementos tratados disminuye respecto a los niveles anteriores. Este porcentaje se sitúa en el 31% para la prueba de 2 hilos, 39% para la prueba de 4 hilos y 25% para la de 8 hilos.

La eficiencia en términos de datos tratados es del 100% para la ejecución con 2 hilos. Para 4 hilos se tiene un 64% de eficiencia. Por último para el caso de 8 hilos se tiene un 20% de eficiencia.

Como se puede observar en todas las ejecuciones se ha llegado al máximo rendimiento posible.

6.1.2.3 PRUEBAS DEL ALGORITMO I PARALELIZADO CON TBB EN CORE I5

A continuación se presentan los resultados obtenidos aplicando las diferentes técnicas de paralelización de TBB al algoritmo I. En esta máquina se ha comprobado que TBB utiliza 4 hilos de ejecución.

Para una carga baja de datos se obtienen los resultados presentados en la Figura 6.28.

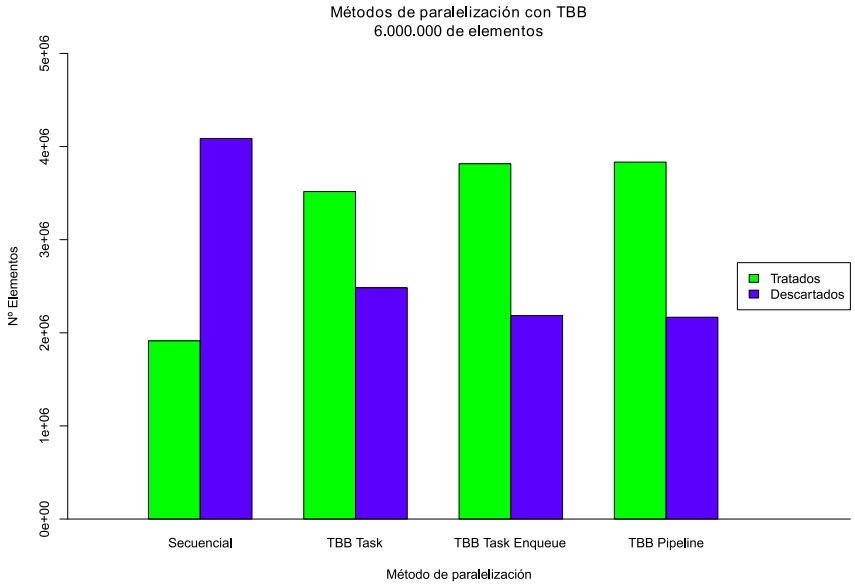

Figura 6.28. Resultado de la ejecución utilizando TBB del algoritmo I con carga baja en Core I5.

En este primer nivel de carga se observa como las tres implementaciones tienen rendimientos cercanos. También se observa como en el caso de hilos de C++ que en este primer nivel de carga se ha llegado al máximo de elementos tratados ya que no se produce aumento para ninguna de las implementaciones entre las 3 pruebas.

El porcentaje de datos tratados en la implementación con *task* es de un 59%. Para la implementación basada en *taskEnqueue* se obtiene un 64%. Por último con la implementación basada en *pipeline* se obtiene un porcentaje de datos tratados del 64%.

La eficiencia es, en la implementación basada en *task* del 45%. En la implementación basada en *taskEnqueue* se alcanza el 50%. Por último la implementación con *pipeline* ofrece una eficiencia del 50%.

Como se puede observar, en esta máquina el resultado de la implementación con *pipeline* no es el que peor rendimiento da, como pasaba en todos los casos de prueba con la

máquina anterior. Esto puede indicar que la tecnología *hyperthreading* produce bloqueos en el *pipeline* y por lo tanto pérdida de rendimiento en algunas etapas.

En la Figura 6.29 se muestra el resultado obtenido para un nivel de carga media.

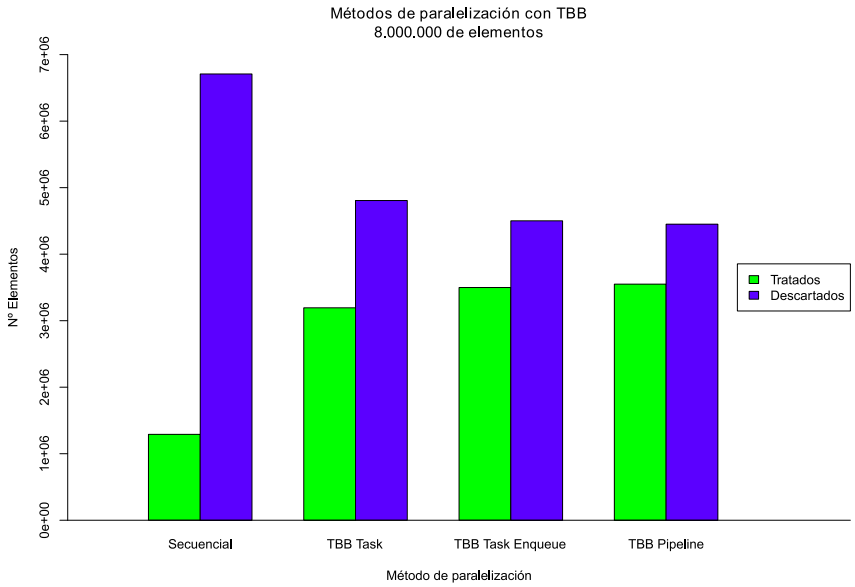

Figura 6.29. Resultado de la ejecución utilizando TBB del algoritmo I con carga media en Core I5.

En este nivel de carga de datos todas las implementaciones han llegado a su límite de tratamiento de datos.

En este nivel de datos se tiene un porcentaje de datos tratados del 40% en la implementación basada en *task*. Para la implementación con *taskEnqueue* se consigue un porcentaje de tratados del 43%. Para la implementación de *pipeline* el porcentaje de datos tratados es del 44%.

Estos porcentajes, sensiblemente más bajos que en el nivel anterior, reafirman el hecho de que todas las implementaciones han llegado a su máximo de rendimiento.

La eficiencia en cuanto al número de datos tratados es un 62% para la implementación con *task.* La implementación basada en *taskEnqueue* se obtiene una eficiencia cercana al 68%. Por último en la implementación basada en *pipeline* se llega al 69% de eficiencia.

Este incremento de eficiencia respecto del nivel anterior se debe a que la prueba en versión secuencial para la misma carga de datos fue capaz de tratar muy pocos datos.

La Figura 6.30 muestra el resultado par un nivel alto de carga de datos.

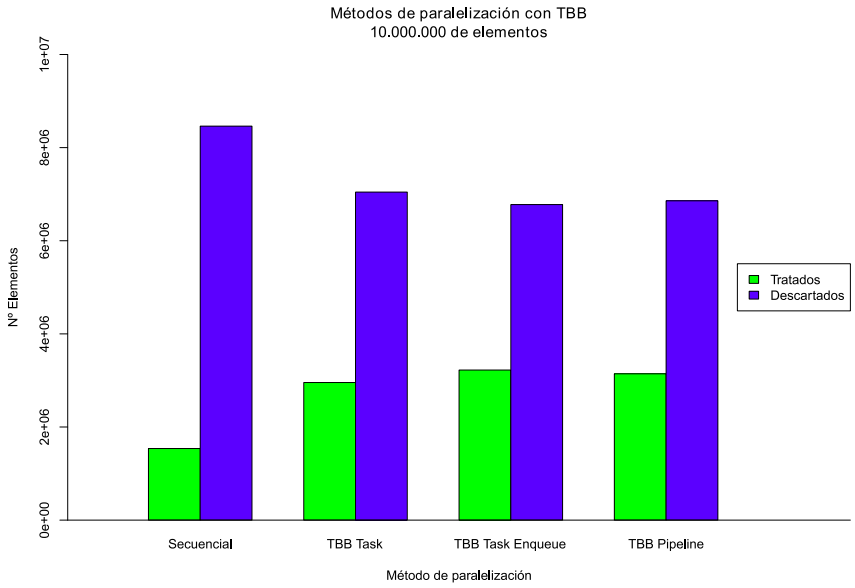

Figura 6.30. Resultado de la ejecución utilizando TBB del algoritmo I con carga alta en Core I5.

En este nivel se observa el mismo comportamiento que en los anteriores. El porcentaje de elementos tratados, en la implementación con *task* se obtiene un 29% de datos tratados. En la implementación con *taskEnqueue* se tiene un porcentaje de tratados del 32%. Para la última implementación, *pipeline*, se obtiene un porcentaje de tratados del 31%.

La eficiencia para este nivel es un 48% para el caso de *task*. En el caso de *taskEnqueue* se obtiene una eficiencia del 52%. Por último en el caso *pipeline* se tiene una eficiencia del 51%.

En general la implementación con TBB produce ganancias en cuanto al número de datos tratados que van desde el 45% en el caso de un nivel de carga baja al 24%. Sin embargo como se lleva viendo en todas las pruebas realizadas el rendimiento es peor que con el modelo de hilos de C++, debido a la sobrecarga que introduce el planificador. La Tabla 6.10 muestra los resultados de datos tratados para los mejores casos de C++ y de TBB.

Carga de datos	Secuencial	Hilos C++ (4)	TBB *TaskEnqueue*
6.000.000	25%	70%	64%
8.000.000	20%	53%	43%
10.000.000	15%	39%	32%

Tabla 6.10. Porcentaje de datos tratados en los mejores casos de cada prueba para algoritmo I en I5.

En cuanto a la eficiencia se tienen que TBB tiene mejores resultado que C++, excepto para el nivel de carga de datos medio, el cual parece que se ajusta en tamaño del problema a la capacidad de cómputo y obtiene una eficiencia muy alta con el modelo de hilos de C++. La Tabla 6.11 muestra los resultados de eficiencia en cuanto a datos tratados de los mejores casos de ambos modelos.

Carga de datos	Hilos C++ (4)	TBB(*taskEnqueue*)
6.000.000	55%	64%
8.000.000	82%	68%
10.000.000	39%	52%

Tabla 6.11. Eficiencia en datos tratados para los mejores casos del algoritmo I en I5.

6.1.2.4 PRUEBAS DEL ALGORITMO II EN VERSIÓN SECUENCIAL EN CORE I5

En este punto se presentan los resultados obtenido en las pruebas del algoritmo II en la versión secuencial ejecutada en una máquina I5.

En la Figura 6.31 se muestra el resultado de la ejecución secuencial para una carga de datos baja.

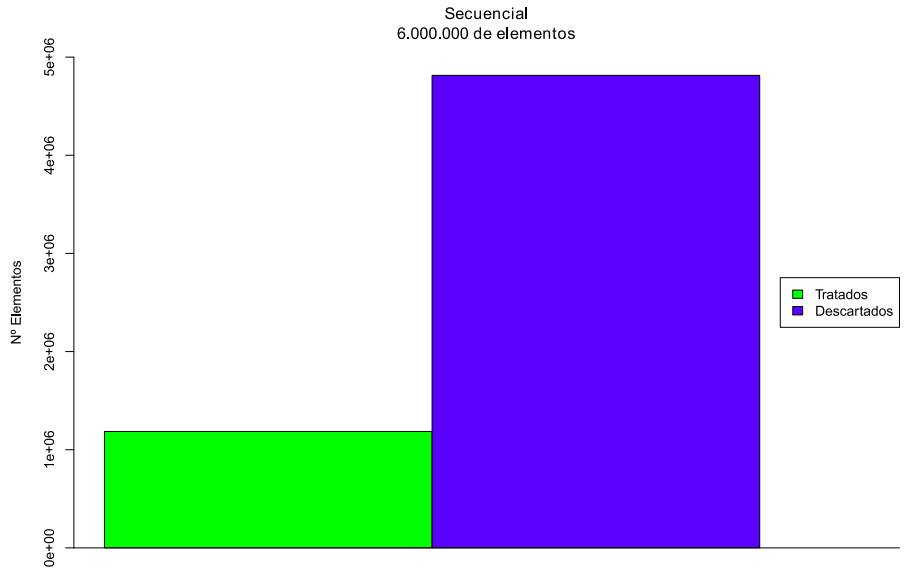

Figura 6.31. Resultado de la ejecución secuencial del algoritmo II con carga baja en Core I5.

En la gráfica se muestra, como en pruebas anteriores, que la ejecución secuencial produce que se descarten muchos datos. En esta prueba se han tratado solamente un 20% de los datos.

Este resultado, para este primer nivel de carga de datos, representa el bajo rendimiento que aporta esta opción por no aprovechar el procesador al máximo. Como se verá en las siguientes pruebas esta tendencia continúa para el resto de niveles de datos.

En la Figura 6.32 se muestra el resultado de la versión secuencial del algoritmo para un nivel de datos medio.

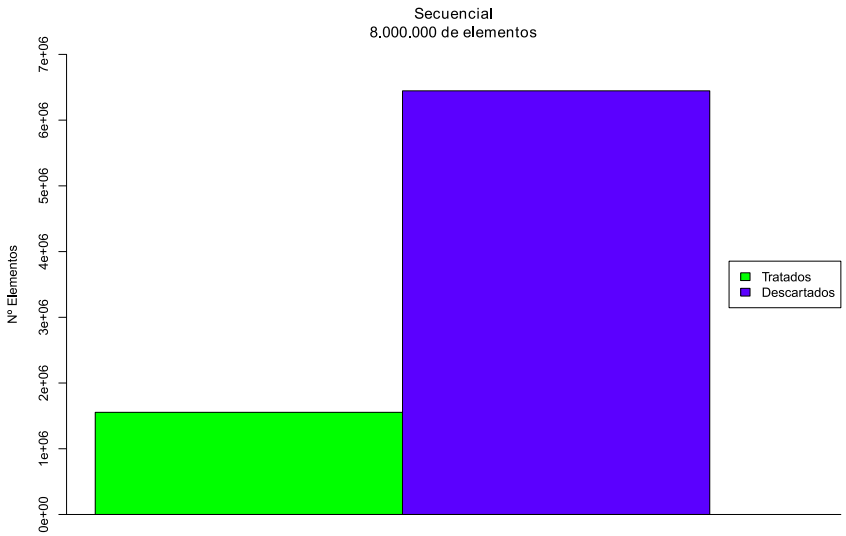

Figura 6.32. Resultado de la ejecución secuencial del algoritmo II con carga media en Core I5.

En esta prueba se observa como se agrava el resultado anterior y se descartan muchos más datos perdiendose más del 80% de los mismos, lo que supone casi 6,5 millones de datos perdidos.

En la siguiente gráfica se muestra el resultado para una carga alta de datos.

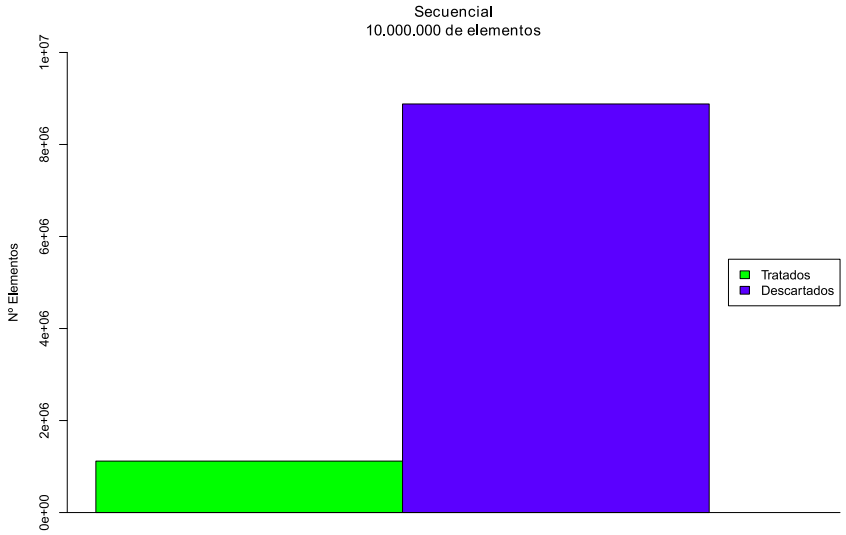

Figura 6.33. Resultado de la ejecución secuencial del algoritmo II con carga alta en Core I5.

En esta prueba se observa un comportamiento similar a las anteriores, con excepción de que el número de datos tratados desciende respecto a ellas. El porcentaje de datos tratados en este nivel de carga se reduce hasta llegar al 11%.

En general para este modo secuencial se observa el mismo comportamiento que se dio en las pruebas con el I7. Debido a que el descarte esta implementado en el hilo de tratamiento y a que cuanta más carga de datos exista mayor será este descarte, el tiempo disponible para tratar datos es menor. Esta situación produce un gran pérdida de rendimiento en el modo secuencial, sin embargo produce que en las versiones paralelas se traten gran cantidad de datos, ya que al disminuir el retraso disminuye, por un lado el número de elementos descartados y, además, esto significa un aumento en el tiempo que se puede dedicar a tratar datos.

Debido a la propia naturaleza del algoritmo el porcentaje de datos tratados siempre es menor que en el caso del algoritmo I, como indica la Tabla 6.12.

Carga de datos	Secuencial Algoritmo I	Secuencial Algoritmo II
6.000.000	25%	20%
8.000.000	19%	18%
10.000.000	15%	11%

Tabla 6.12. Elementos tratados en versión secuencial de algoritmos I y II en un I5.

6.1.2.5 PRUEBAS DEL ALGORITMO II PARALELIZADO CON MODELO DE HILOS DE C++ EN CORE I5

En este punto se presentan los resultados obtenidos en las pruebas del algoritmo II con el modelo de hilos de C++.

En la siguiente figura se muestra el resultado para una carga de datos baja.

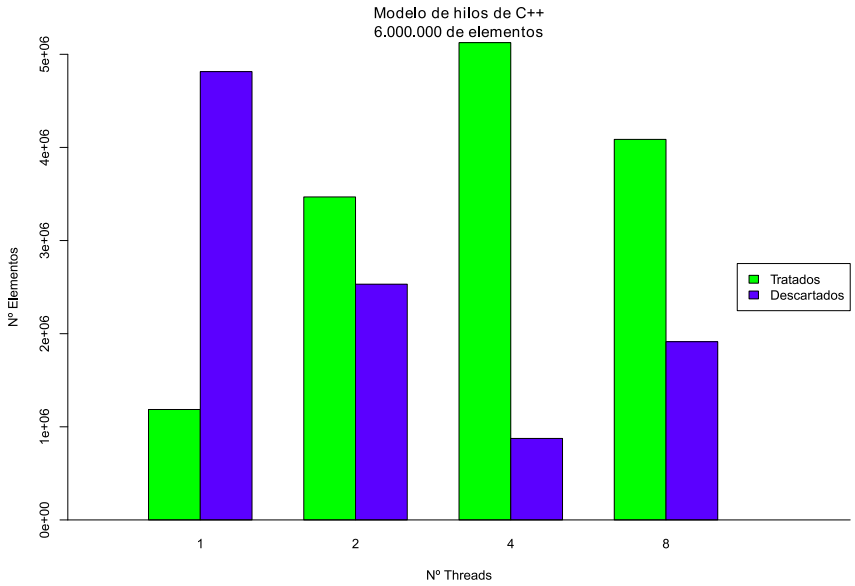

Figura 6.34. Resultado de la ejecución con hilos de C++ del algoritmo II con carga baja en Core I5.

Para este nivel de datos la paralelización con hilos de C++ supone una gran ganancia en cuanto a datos tratados. Para la paralelización 2 hilos se obtiene un porcentaje de datos tratados de cerca del 58%, un 38% más que ne la versión secuencial. En el caso de 4 hilos se consigue el máximo porcentaje para este nivel, un 85%. En la paralelización con 8 hilos se obtienen mejores resultados que con la de 2, llegando a un porcentaje del 68%.

La eficiencia, en cuanto a datos tratados, en este nivel de datos es muy alta, llegando al 146% en el caso de 2 hilos. Para el caso de 4 hilos se obtiene un valor del 108%. Por último en el caso de 8 hilos este valor se sitúa en el 43%.

Estos valores muestran la alta escalabilidad del algoritmo en este nivel de datos y la eficiencia del mismo cuando se trabaja con varios hilos, siendo, en los caso de 2 y 4 hilos, cada uno de ellos de tratar más datos que con uno solo.

En la Figura 6.35 se muestra el resultado para una carga media de datos.

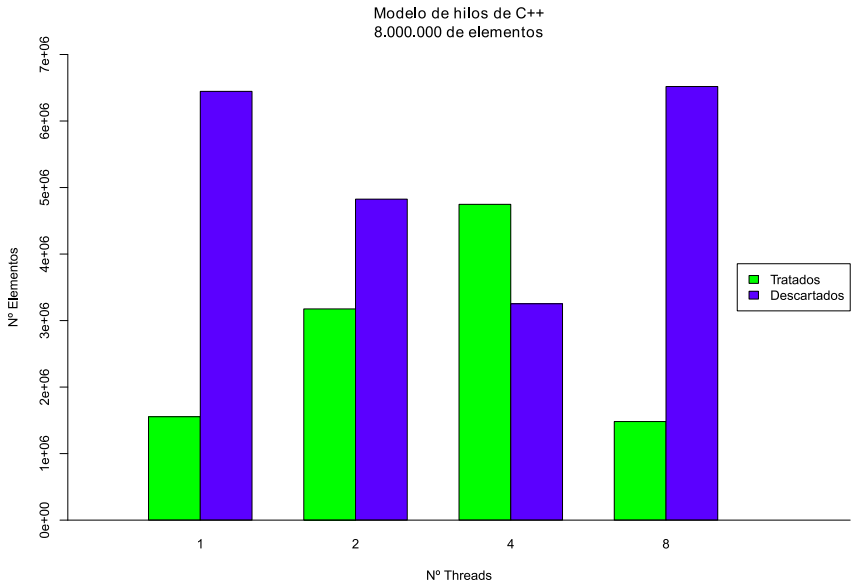

Figura 6.35. Resultado de la ejecución con hilos de C++ del algoritmo II con carga media en Core I5.

Para este nivel de datos se observa que el número total de datos tratados de las versiones con 4 y 8 hilos disminuye respecto al nivel anterior. Esto indica que el algoritmo necesita, en proporción, descartar más datos para mantener el mismo valor de retraso, lo que quiere decir que el algoritmo en esta máquina no es escalable en cuanto al número de datos y disminuye su rendimiento al aumentar el número de datos. Esto tiene su explicación en que el descarte se hace en los propios hilos de tratamiento y por tanto, cuanto más datos haya que descartar más tiempo se tardará y menos datos se tratarán.

El porcentaje de datos tratados en este nivel es del 40% en la prueba con 2 hilos. Para la prueba con 4 hilos se tiene un porcentaje de datos tratados del 59%. Para la implementación con 8 hilos se produce una pérdida de rendimiento muy alta y el número de datos descartados disminuye hasta el 19%, quedandose a niveles de versión secuencial.

La eficiencia en cuanto a datos tratados de la versión con 2 hilos se encuentra en el 102%. En la versión con 4 hilos se tiene una eficiencia en cuantoa datos tratados del 76%. En la versión con 8 hilos se tiene una eficiencia del 12%.

En la Figura 6.36 se muestra el resultado de la prueba con una carga de datos alta.

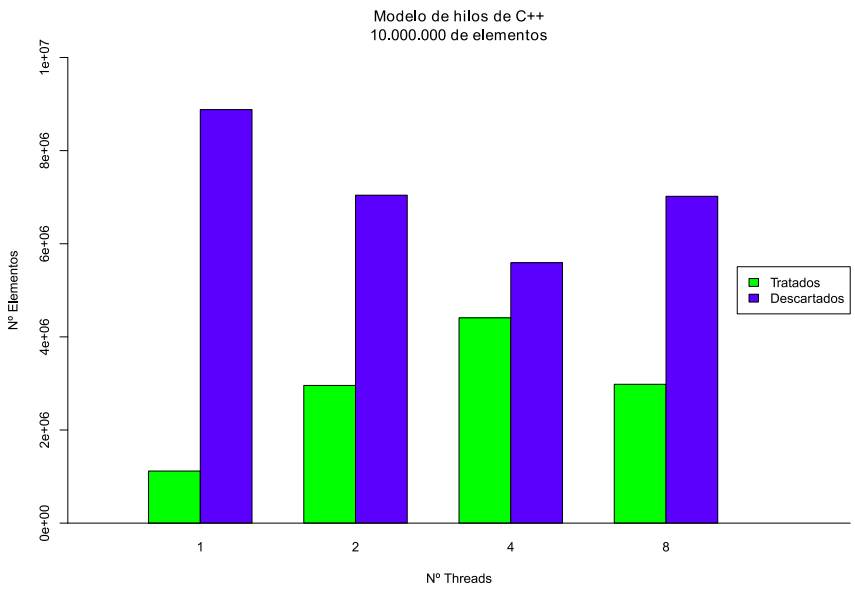

Figura 6.36. Resultado de la ejecución con hilos de C++ del algoritmo II con carga alta en Core I5.

En esta última prueba se observa el mismo comportamiento que en la anterior. Se tratan menos datos en cada prueba debido al mayor descarte, hasta el punto de llegar a haber más descarte que elementos tratados.

El porcentaje de datos de datos tratados en el caso de 2 hilos para este nivel de carga es del 29%. Para la versión de 4 hilos se llega a tratar el 44% de los datos y en la de 8 hilos se queda en el 29%.

La eficiencia de este nivel no disminuye al ritmo que el porcentaje de datos tratados ya que la versión secuencial del algoritmo ha tratado muy pocos datos, menos que en los 2 niveles anteriores. El valor de la eficiencia, en cuanto a datos tratados, es del 130% en el caso de 2 hilos. Para el caso de 4 hilos este valor se sitúa en el 98%. Por último, para el caso de 8 hilos este valor es del 33%.

Con esta paralelización se ha conseguido aumentar el número de datos tratados en cada nivel de carga de forma notable. También se ha conseguido obtener una gran eficiencia, en cuanto a datos tratados, al reducir el retraso, como ya se explico en las pruebas secuenciales.

En resumen se ha conseguido un incremento en datos tratados del 50% en el nivel de carga bajo, 35% en el medio y 28% en el nivel de carga alto.

La implementación con este algoritmo ha conseguido unos porcentajes de tratamiento de datos mayores que en el caso del algoritmo I. Los resultados de ambos para 4 hilos se resumen en la Tabla 6.13.

Carga de datos	Hilos C++(4) Algoritmo I	Hilos C++(4) Algoritmo II
6.000.000	70%	85%
8.000.000	53%	59%
10.000.000	39%	44%

Tabla 6.13. Comparativa de datos tratados por hilos de C++ en algoritmos I y II en I5.

La razón de la obtención de estos resultados es la forma de trabajar de ambos algoritmos. Como ya se ha dicho, el algoritmo I descarta gran cantidad de datos que a priori se podrían tratar. El algoritmo II, sin embargo, ajusta mucho más el descarte a los datos que realmente no se pueden tratar. Además, como ya se ha explicado, el hecho de aumentar la capacidad de cómputo y por tanto reducir el retraso del sistema en el algoritmo II, hace que el descarte sea menor y esto a su vez deja más recursos para tratar datos, ya que tanto descarte como tratamiento de datos son llevados a cabo por los mismos hilos.

6.1.2.6 PRUEBAS DEL ALGORITMO II PARALELIZADO CON TBB EN CORE I5

En este punto se muestran los resultados obtenidos en la ejecución de las pruebas con las diferentes implementaciones de TBB.

En la Figura 6.37 se muestra el resultado para un nivel bajo de carga de datos.

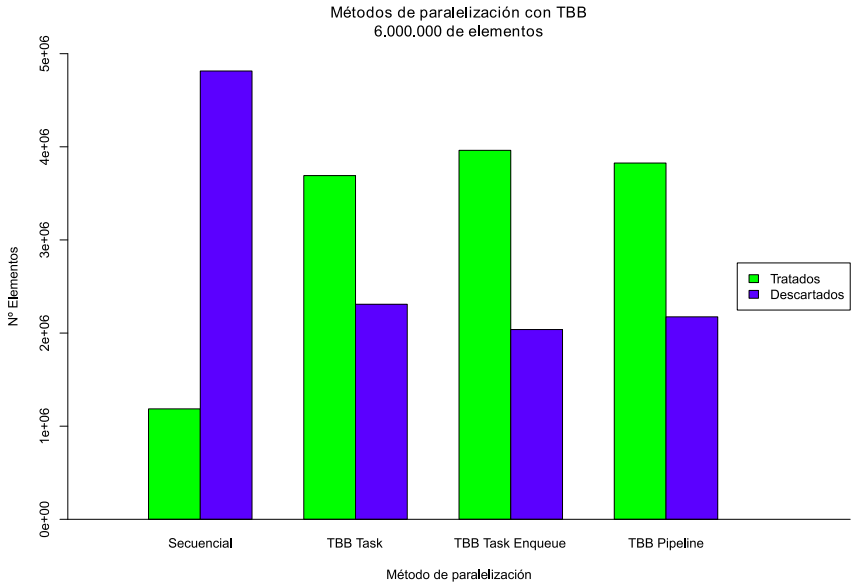

Figura 6.37. Resultado de la ejecución utilizando TBB del algoritmo II con carga baja en Core I5.

Como se puede observar las 3 implementaciones obtienen resultado similares, aunque destaca ligeramente la implementación basada en *taskEnqueue*, con un porcentaje de datos tratados del 66%. El porcentaje de datos tratados para la implementación con *task* es del 61%. Por último la implementación basada en *pipeline* consigue un porcentaje del 63% de los datos totales.

La eficiencia en cuanto a datos tratados es del 77% para la implementación con *task.* Para la implementación con *taskEnqueue* la eficiencia es del 83%. Para la última implementación, *pipeline*, la eficiencia es del 80%.

La siguiente figura muestra los resultados para un nivel de carga medio

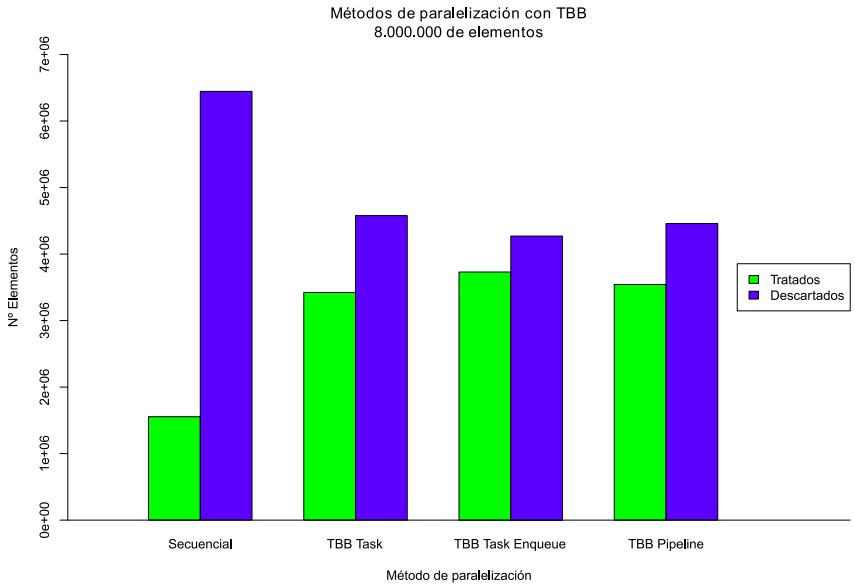

Figura 6.38. Resultado de la ejecución utilizando TBB del algoritmo II con carga media en Core I5.

En este caso se observa que todas la implementaciones han llegado a su máximo, ya que el número de datos tratados en cada una de ellas es el mismo que en el nivel de carga anterior. Sin embargo, debido al aumento de datos totales, se ha reducido el porcentaje de datos tratados quedando en el 42% en la implementación con *task*, el 46% con *taskEnqueue* y el 44% en la versión con *pipeline*.

La eficiencia en cuanto a datos tratados es en el caso de *task* es del 55%. Para la versión con *taskEnqueue* se tiene un valor de eficiencia del 60%. En el caso de *pipeline* se tiene una eficiencia del 57%.

En la Figura 6.39 se muestra el resultado de la pruebas con una carga alta de datos.

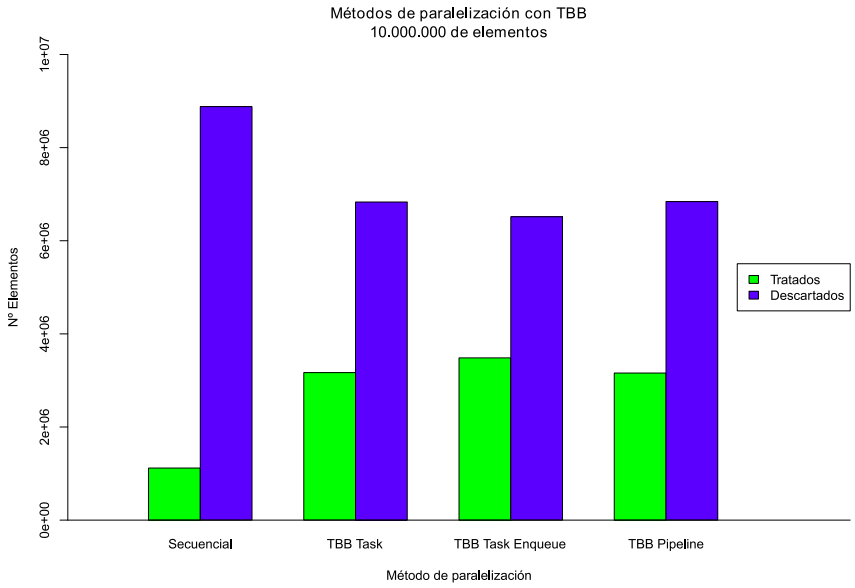

Figura 6.39. Resultado de la ejecución utilizando TBB del algoritmo II con carga alta en Core I5.

Esta prueba muestra un comportamiento similar al de la anterior, todas las implementaciones ha llegado a su máximo nivel de rendimiento y el porcentaje de descartados aumenta.

Para este nivel de datos se tiene un porcentaje de datos tratados en la implementación con *task* del 31%, la implementación con *taskEnqueue* llega al 34% de los datos totales y, por último la implementación basada en *pipeline* trata el 31% de los datos.

La eficiencia en cuanto a datos tratados es del 70% en la implementación con *task.* En el caso de *taskEnqueue* del 78%. Por último en el caso de *pipeline* es del 70%.

En líneas generales la implementación del paralelismo con TBB produce una ganancia alta en cuanto a la cantidad de datos tratados. Al igual que en casos anteriores la implementación con *taskEnqueue* es la que más rendimiento produce de las tres.

La implementación del algoritmo II con TBB produce resultados muy similares a la implementación del algoritmo I, como se indica en la Tabla 6.14.

Carga de datos	TBB(*taskEnqueue*) Algoritmo I	TBB(*taskEnqueue*) Algoritmo II
6.000.000	64%	63%
8.000.000	43%	43%
10.000.000	32%	34%

Tabla 6.14. Comparativa de datos tratados con TBB en algoritmos I y II en I5.

En cuanto al rendimiento respecto a la implementación con el modelo de hilos de C++ en la se observa como el mejor caso de TBB queda por debajo del mejor caso de C++ en todos los niveles de carga de datos. Este resultado es debido a la sobrecarga que produce el planificador de TBB.

Carga de datos	Secuencial	Hilos C++ (4)	TBB *TaskEnqueue*
6.000.000	20%	85%	63%
8.000.000	18%	59%	43%
10.000.000	11%	44%	34%

Tabla 6.15. Porcentaje de datos tratados en los mejores casos de cada prueba para algoritmo II en I5.

En cuanto a la eficiencia, la Tabla 6.16 muestra como el modelo de hilos es mucho más eficiente en cuanto al tratamiento de datos que TBB. Esta mayor eficiencia del modelo de C++ viene dada por la misma razón por la que es capaz de tratar más datos, no ejecuta un planificador para decidir qué tarea ejecuta en cada momento.

Carga de datos	Hilos C++ (4)	TBB(*taskEnqueue*)
6.000.000	108%	83%
8.000.000	76%	60%
10.000.000	98%	78%

Tabla 6.16. Eficiencia en datos tratados para los mejores casos del algoritmo II en I5.

6.1.2.7 PRUEBAS DEL ALGORITMO III EN CORE I5

Al igual que ocurría en la anterior máquina la cantidad de memoria necesaria para realizar las pruebas de este algoritmo es muy alta. Este problema se agrava en esta máquina que cuenta con la mitad de memoria que la anterior, 4GB.

Con 4GB de memoria en el mejor de los casos es imposible ni siquiera introducir los datos que no caben en memoria principal en la memoria virtual, ya que esta sólo tiene 4GB de espacio. Esto hace que la suma de ambas sea insuficiente para poder albergar todos los datos de la simulación, además de aquellos que necesite el sistema operativo, siendo imposible ejecutar las pruebas en esta máquina.

6.1.3 PRUEBAS DE RENDIMIENTO EN ARQUITECTURA CC-NUMA

La máquina CC-NUMA donde se ha realizado las pruebas está compuesta por 4 *sockets*. Cada uno de estos *socket* tiene un procesador Intel XEON con 6 cores a 1,87GHz. Esto hace un total de 24 núcleos, que además cuentan con tecnología *hyperthreading*, lo que permite ejecutar hasta 48 hilos simultáneamente.

La memoria total de la máquina donde se han realizado las pruebas es de 128GB, divida en 4 bancos de 32GB, uno por *socket*. Cabe destacar que el acceso a cada uno de estos bancos tiene un coste distinto dependiendo del *socket* desde el que se acceda. El acceso de un *socket* a su banco es mucho más barato, en términos temporales, que el acceso a un banco externo. Las especificaciones detalladas se encuentran en el Anexo I de este documento.

Cada core de la máquina cuenta con su propia caché tanto L1 como L2, y cada *socket* cuenta con una caché L3 compartida por todos los cores del chip. Dado que estamos ante una arquitectura CC-NUMA se mantiene la coherencia de caché, no sólo dentro de un propio chip sino entre todos ellos. La ventaja de mantener la coherencia permite que los procesos puedan tener una visión coherente de toda la memoria del sistema. Sin embargo si el acceso a memoria de los diferentes hilos de un mismo proceso no mantiene localidad espacial ni temporal el rendimiento de este tipo de arquitecturas disminuye drásticamente.

La implementación que se ha llevado a cabo no cumple estos criterios de localidad espacial y temporal ya que las diferentes colas de datos no se asocian a ningún hilo, sino que, en cada iteración del algoritmo cada hilo tratará el dato de una cola distinta. Por otra parte los hilos no están asociados a ningún core concreto, sino que se moverán por ellos según decida el S.O. Ambas cosas producen que cuando un hilo toma un dato de una cola, muy probablemente éste no estará en ninguno de los 3 niveles de caché del chip donde ejecute. Además, es muy probable que el dato requerido se encuentre en la caché de otro chip y que se encuentre modificada, es decir, no actualizada en la memoria principal, por lo que se deberá volcar desde la caché del chip en cuestión al banco de memoria asociado a ese chip, y de ahí, al banco de memoria del chip donde se ejecute el hilo que requiere el dato.

Las siguientes figuras muestran el resultado del algoritmo I para una carga media de datos. La Figura 6.40 muestra el resultado de la ejecución con el modelo de hilos de C++. Se puede observar como el rendimiento es bajísimo en todas las configuraciones de hilos, quedando a niveles de implementación secuencial.

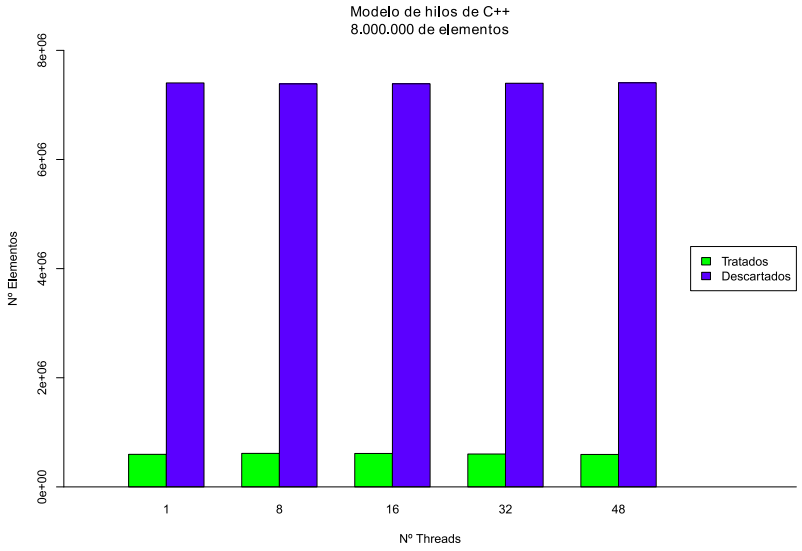

Figura 6.40. Ejecución de la implementación con C++ en arquitectura CC-NUMA.

A continuación la Figura 6.41 muestra el resultado de la ejecución con las diferentes implementaciones en TBB.

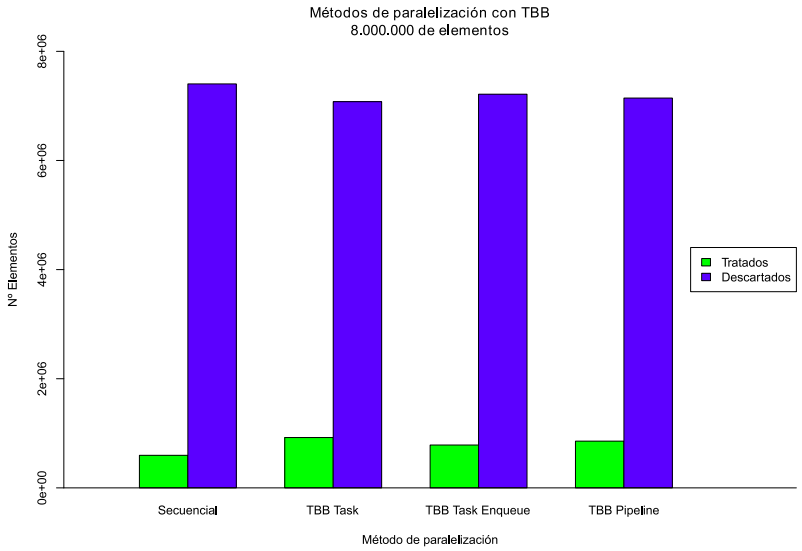

Figura 6.41. Ejecución de la implementación con TBB en arquitectura CC-NUMA.

En este caso se observa el mismo comportamiento que en el anterior, el rendimiento es muy bajo en las 3 implementaciones.

El motivo de este rendimiento tan bajo en las diferentes implementaciones llevada a cabo está en la cantidad de fallos de caché. Se ha medido el porcentaje de fallo de caché en la distintas ejecuciones del programa y se ha obtenido un 92% de media de fallo de caché.

Para llevar a cabo estas mediciones se ha utilizado la herramienta *likwid*. *Likwid* es una herramienta que permite acceder a los contadores hardware del procesador y obtener datos relativos a la ejecución de los programas, como fallos de caché, predicción de saltos, etc.

Likwid está formado por diferentes programas que permiten llevar a cabo diferentes análisis, para llevar a cabo las mediciones de fallo de caché se ha utilizado el programa *likwid-perfctr.* La ejecución de esta prueba se hace a través de la siguiente línea:

```
likwid-perfctr -c 0-47 -g CACHE -o pruebaCache.txt ./<programa a ejecutar>
```

Esta línea ejecuta *likwid-perfctr* con os siguientes parámetros:

- -c: Este parámetro indica los procesadores sobre los que se va a efectuar la medición. Se puede indicar como procesadores aislados, nodos completos o intervalos, en este caso se indica que la medición se realizará en los procesadores entre 0 y 47, es decir, todos los procesadores (lógicos) de la máquina.
- -g: Este parámetro indica los grupos de datos que se quieren medir. Dependiendo de la máquina estos grupos varían, ya que cada procesador tiene sus propios contadores. Para comprobar que grupos de medición existen en la máquina que se vaya a utilizar se puede ejecutar *likwid-perfctr* con la opción –a. En este caso se ha utilizado el grupo de medición CACHE, que ofrece los datos de tasas de fallo y acierto de las diferentes cachés del sistema.
- -o: Con este parámetro se guarda el resultado en un fichero de salida, en este caso pruebaCache.txt
- El último parámetro indica el programa que se va a ejecutar.

Esta prueba se ha realizado en las 3 máquinas en que se ha llevado a cabo pruebas. El resultado es el recogido en la Tabla 6.17.

Máquina de pruebas	Tasa de fallo en caché
Intel Core I5	18%
Intel Core I7	15%
Intel XEON (CC-NUMA)	94%

Tabla 6.17. Tasa de fallos de caché en las diferentes máquinas utilizadas.

Como se puede observar la tasa de fallos de caché en la arquitectura CC-NUMA es muy alta respecto a las otras dos. La única forma de solucionar este inconveniente sería modificar el código para llevar a cabo un balanceo de carga y que cada hilo trate los datos de un subgrupo de colas específico.

6.2 PRUEBA DE PRECISIÓN

En este punto se muestra el resultado de las pruebas de precisión llevadas a cabo. La prueba que se ha llevado a cabo ha sido calcular el error relativo que se produce al realizar un número concreto de predicciones por cada dato real de corrección. El número de predicciones por corrección que se han llevado a cabo han sido 2, 4, 6, 8 y 10. Para realizar la prueba se ha tomado un objeto móvil cuya trayectoria es un hipódromo de aproximación. Este movimiento es el que realizan los aviones cuando están esperando permiso para aterrizar.

La forma de llevar a cabo la prueba ha sido la siguiente:

- En cada ejecución se decide un número, *n*, de predicciones por cada corrección.
- Durante la ejecución se realizan predicciones en todas las iteraciones, pero solo se corrige con la medida real cada *n* iteraciones, ignorando el resto de medidas reales.
- El error se calcula tras la etapa de predicción de cada iteración, comparando con la medida real que corresponda en ese momento aunque ésta no se vaya a utilizar para corregir posteriormente.

La Figura 6.42 muestra el resultado de la prueba.

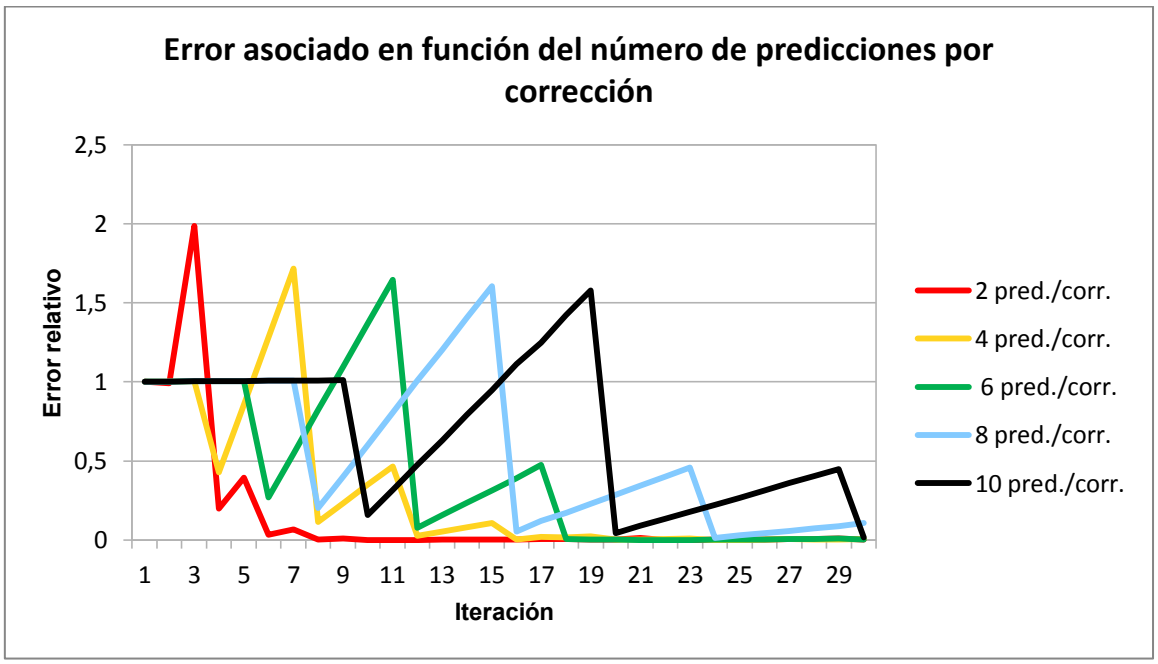

Figura 6.42. Resultado de la prueba de precisión.

El gráfico muestra que cuantas más correcciones se realizan, antes converge el algoritmo y antes disminuye el error. Como se puede observar para cada caso el error se mantiene en 1 hasta que llega la primera corrección, momento en que disminuye rápidamente, y que coincide con el número de predicciones por cada corrección en cada uno de los casos. Posteriormente el error aumenta en cada iteración, hasta que llega otra corrección, momento en que el error vuelve a disminuir. Este comportamiento se produce continuamente y tras cada corrección el error aumenta menos, tendiendo al mínimo valor que es cercano a 0. Esto se produce antes cuando la relación de correcciones por predicción es mayor.

7 CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se detallan las conclusiones extraídas y se proponen trabajos futuros relacionados con el proyecto.

7.1 CONCLUSIONES

En este proyecto se ha llevado a cabo la paralelización del algoritmo del filtro de Kalman lineal o discreto con diferentes técnicas. Además, se ha implementado bajo el prisma de un sistema de tiempo real de calidad de servicio, que debe llevar a cabo las operaciones sin acumular un retraso temporal excesivo.

Las conclusiones se van a dividir en 3 grupos. En el primer grupo se abordará la implementación secuencial del algoritmo del filtro de Kalman, la gestión de los datos y la simulación de llegada de los datos. En el segundo se hablará sobre los algoritmos de descarte que se han implementado para mantener el retraso controlado. En el tercer y último grupo se tratarán las diferentes técnicas de paralelización que se han utilizado.

Con la implementación secuencial del algoritmo se ha conseguido dotar al sistema de la funcionalidad básica para llevar a cabo los cálculos necesarios. Además se ha realizado una implementación optimizada utilizando las facilidades que incluye el último estándar de C++. En cuanto al sistema gestor de datos, se ha conseguido implementar un sistema capaz de almacenar los datos que se reciben y de servirlos de forma ordenada y equitativa. Este sistema se ha implementado de forma que pueda soportar el acceso concurrente por varios hilos minimizando el número y tiempo de bloqueos.

En esta fase se ha implementado un sistema en tiempo real que permite simular la llegada de los datos desde los sensores. Este simulador inserta los datos en el sistema gestor según su marca de tiempo, permitiendo aproximarse a un caso real con sensores.

Las pruebas de rendimiento que se han llevado a cabo con esta implementación muestran como su rendimiento es muy bajo para las cantidades de datos que se han propuesto. El hecho de obtener un rendimiento tan bajo en esta versión tras optimizarla motiva la paralelización de las operaciones algebraicas y de las tareas que se realizan.

Con esta primera parte del proyecto se ha conseguido disponer de la funcionalidad básica en operaciones matemáticas, implementación del algoritmo del filtro de Kalman

discreto y gestión de datos de medida y de resultados, asentando además, las bases para la posterior paralelización del sistema.

La segunda parte del proyecto se ha centrado en la implementación de algoritmos que permitan descartar los datos que no se puedan tratar. Se han implementado tres algoritmos, el primero de ellos basa su funcionamiento en el vaciado de las colas cada cierto tiempo. Este vaciado asegura que el máximo retraso que se tendrá en el tratamiento de datos será el periodo de vaciado de las colas.

Uno de los problemas que presenta este algoritmo es que necesita un hilo propio para su ejecución, obligando a sincronizarlo con el resto, lo que produce una mayor complejidad en el sistema. También tiene la desventaja de que al ser un hilo con requerimientos temporales, puede ocurrir que en ocasiones no se ejecute a tiempo, debido a que estamos en un sistema de calidad de servicio, produciendo un retraso general en el sistema. Otro problema que presenta es que en ocasiones puede descartar más datos de los necesarios, sobre todo si su ejecución coincide con la ejecución del hilo de inserción de datos.

En cuanto a las ventajas destaca la facilidad de implementación del algoritmo; la velocidad en cuanto a las operaciones de memoria del mismo, ya que sólo realiza una operación que libera la memoria ocupada por los datos contenidos en las colas y que al ejecutarse en un hilo independiente no “roba” tiempo a los hilos de tratamiento de datos, como el resto de algoritmos.

El segundo algoritmo basa su funcionamiento en realizar el descarte calculando el retraso que tiene cada elemento que se trata. La decisión del número de elementos descartados y los valores de retraso en los que se aplica cada cantidad de descarte se ha tomado en base a resultados experimentales, de forma que se descartase el menor número de datos posible manteniendo el retraso en el menor nivel de los definidos, es decir, por debajo de los 250 ms.

Los mayores inconvenientes que presenta este algoritmo es que se ejecuta en el mismo hilo que el tratamiento de datos, por lo tanto, toma parte del tiempo del que disponen estos hilos provocando que disminuya el tiempo disponible para el tratamiento. Otro de los inconvenientes es que, debido a las limitaciones de los contenedores utilizados para implementar las colas de datos, para llevar a cabo el descarte es necesario extraer el dato de la cola en cuestión, lo que conlleva movimiento de datos en memoria y por lo tanto, sobre todo

en el caso de que parte de los datos no estén en caché lleva bastante más tiempo que liberarla directamente.

En cuanto a las ventajas destaca que este algoritmo se ajusta mucho más al estado del sistema en cada momento, evitando en gran medida el descarte de datos que aún podían tratarse que se producía en el algoritmo anterior. Además el descarte que realiza este algoritmo no produce el problema que producía el anterior al ejecutar a la vez que la inserción de datos.

El tercer algoritmo implementado basa su funcionamiento en la capacidad de tratamiento del sistema. La idea de este algoritmo es calcular el número aproximado de datos que es capaz de tratar el sistema en base al número de datos tratados anteriormente.

Este algoritmo necesita un hilo dedicado para su ejecución que se encarga de tomar los valores de datos tratados cada cierto tiempo. Este hilo debe cumplir restricciones temporales ya que su retraso provoca que el conteo de datos para ese intervalo de tiempo sea incorrecto, aunque esto no produce el fallo del sistema, sólo produce un aumento del retraso debido a que se intente tratar más datos de los posibles. La ejecución de este hilo dedicado no interfiere con el resto en la misma medida que lo hacía en el caso del primer algoritmo, ya que no accede en ningún momento a las colas de datos. Además de este hilo el algoritmo ejecuta parte de su funcionalidad en los hilos de tratamiento, donde calcula en base al histórico calculado y al tamaño de cada cola cuantos elementos de cada objeto móvil ha de descartar. Esto hace que el tiempo disponible en los hilos de tratamiento deba repartirse entre el propio tratamiento y parte del descarte. Otro de los inconvenientes, al igual que el algoritmo anterior, es que debido a las limitaciones del contenedor que implementa las colas para realizar el descarte es necesario mover el dato a descartar en memoria, siendo esta última operación más costosa que la eliminación directa.

En cuanto a las ventajas de este algoritmo, la más destacable, es que además de ajustarse al estado y capacidad del sistema se ajusta al número de datos que existen del objeto sobre el que se va a realizar el descarte, evitando, en el caso de objeto con pocas medidas, que se descarten la mayoría de ellas, ya que el descarte afecta más a la precisión de estos objetos.

En líneas generales y viendo el resultado de las pruebas el algoritmo más recomendable es el segundo ya que se ajusta mejor, en este caso al retraso del sistema. Debe tenerse en cuenta que debido a la cantidad de memoria necesaria para la simulación no ha

sido posible probar el tercer algoritmo, aunque en principio parece que este sería el más justo en todos los ámbitos y por lo tanto el más recomendable.

La última fase ha sido la paralelización del algoritmo. La paralelización se ha divido en dos partes, la primera ha sido una paralelización de las operaciones algebraicas más pesadas computacionalmente, que en este caso ha resultado ser el producto de matrices. La segunda parte de la paralelización ha sido paralelizar las tareas de forma que se traten datos de varios objetos distintos de forma concurrente.

En el caso de la primera fase las pruebas realizadas muestran que aunque las dos técnicas utilizadas aceleran le ejecución en casi todos los casos no son útiles en este caso debido al pequeño tamaño de las matrices.

En la segunda fase es donde se han conseguido muy buenos resultados en todas las implementaciones, tanto con el modelo de hilos de C++ como con las diferentes implementaciones con TBB. Las pruebas realizadas muestran como se ha obtenido un gran aumento de rendimiento y una buena escalabilidad en todas las implementaciones y para todos los algoritmo que han podido ser sometidos a prueba en las dos máquinas UMA en las que se probado.

Por otra parte la prueba en la arquitectura CC-NUMA muestra que la implementación no es buena para su ejecución en este tipo de máquina y que por tanto deben hacerse cambios en las implementaciones para adaptarlas a este tipo de arquitecturas, lo cual se deja como trabajo futuro.

En líneas generales se ha conseguido cumplir los objetivos marcados al comienzo del proyecto y desarrollar un sistema que cumple los requisitos impuestos.

7.2 TRABAJOS FUTUROS

Los trabajos futuros que se pueden llevar a cabo partiendo de este proyecto se dividen en dos grupos. Por un lado la implementación de las diferentes versiones del filtro de Kalman y por otra la ampliación en las técnicas y métodos de paralelización, así como la optimización para trabajar en diferentes arquitecturas como NUMA.

Dentro del primer se propone implementar un sistema capaz de aplicar las diferentes versiones del filtro de Kalman:

- Filtro de Kalman discreto.
- Filtro de Kalman extendido.
- Filtro de Kalman "*Unscented*".
- Filtro de Kalman-Bucy.

Este sistema permitiría al usuario utilizar la implementación que mejor se ajuste a su sistema.

Por otra parte se propone la implementación del paralelismo haciendo uso de diferentes técnicas a las utilizadas en este proyecto. Entre estas técnicas se destacan 2:

- CUDA: Debido a que CUDA permite realizar cómputo general en tarjetas gráficas y que estas son dispositivos SIMD, es muy recomendable implementar sobre ellas cualquier operación vectorial o matricial.
- OpenCL: Esta librería permite llevara a cabo cómputo paralelo con multitud de dispositivos compatibles, entre ellos tarjetas gráficas. Sería una gran opción al igual que CUDA, además cabe la posibilidad de comparar ambas tecnologías.

También se propone la optimización o adecuación de este proyecto para su utilización en arquitectura NUMA, CC-NUMA y para computación híbrida como CPU+GPU o las nuevas arquitecturas MIC de Intel.

8 BIBLIOGRAFÍA

[1] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Journal of Basic Engineering,* pp. 35-45, 1960.

[2] G. Welch y G. Bishop, «An Introduction to the Kalman Filter,» 2006.

[3] A. Pascual, «EKF y UKF: dos extensiones del Filtro de Kalman para sistemas no lineales aplicadas al control de un péndulo invertido,» 2006.

[4] E. A. W. a. R. v. d. Menve, «The Unscented Kalman Filter for Nonlinear Estimation,» 2000.

[5] W. Stallings, Organización y arquitectura de computadores, Pearson, 2005.

[6] F. B. Pérez, «Implementación y evaluación de la factorización de Cholesky mediante TBB y threads en arquitecturas multicore».

[7] C. Lomont, «Introduction to Intel® Advanced Vector Extensions,» 21 Junio 2011. [En línea]. Available: http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions/. [Último acceso: 19 Abril 2012].

[8] O. A. R. Board, *OpenMP. Application Program Interface. Version 3.1,* 2011.

[9] J. Reinders, Intel Threading Building Blocks.Outfitting C++ for Multi-core Processor Parallelism, O'Reilly, 2007.

[10] Intel Corporation, "Intel Threading Building Blocks," 20 Enero 2012. [Online]. Available: threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference.pdf. [Accessed 2012 Abril 20].

[11] A. D. Pangborn, «Scalable Data Clustering Using GPU's,» New York, 2010.

[12] AMD, «OpenCL,» 2012. [En línea]. Available: http://www.amd.com/es/products/technologies/stream-technology/opencl/pages/opencl-intro.aspx. [Último acceso: 7 Mayo 2012].

[13] «International Organization for Standardization,» 2011. [En línea]. Available: http://www.iso.org/iso/catalogue_detail?csnumber=50372. [Último acceso: 8 Mayo 2012].

[14] «CPP Reference,» 19 Abril 2012. [En línea]. Available: http://en.cppreference.com. [Último acceso: 2012 Mayo 8].

[15] O. G. Lorenzo, J. A. Lorenzo, D. B. Heras, J. C. Pichel y F. F. Rivera, «Herramientas para la monitorización de los accesos a memoria de códigos paralelos mediante contadores

hardware,» Septiembre 2011. [En línea]. Available: http://jp2011.pcg.ull.es/sites/jp2011.pcg.ull.es/files/Herramientas_Monitorizacion_Accesos_Mem.pdf. [Último acceso: 15 Marzo 2012].

[16] Intel Corporation, «Intel Software Network,» Intel Corporation, 2008. [En línea]. Available: http://software.intel.com/file/6737. [Último acceso: 18 Marzo 2012].

[17] R. Gerber, The Software Optimiation Cookbook, Intel Press, 2002.

[18] Microsoft Corporation, «MSDN Library,» [En línea]. Available: http://msdn.microsoft.com/en-us/library/ms364057%28VS.80%29.aspx. [Último acceso: 13 Marzo 2011].

[19] B. Barney, «OpenMP,» Lawrence Livermore National Laboratory, 14 Febrero 2012. [En línea]. Available: https://computing.llnl.gov/tutorials/openMP/. [Último acceso: 20 Febrero 2012].

[20] R. Seacord, «Concurrency, Visibility, and Memory,» Carnegie Mellon, 1 Noviembre 2011. [En línea]. Available: https://www.securecoding.cert.org/confluence/display/java/Concurrency,+Visibility,+and+Memory. [Último acceso: 20 Febrero 2012].

ANEXO I – CARACTERÍSTICAS DE LAS MÁQUINAS DE PRUEBAS

En este anexo se detallan las características técnicas de las máquinas utilizadas para llevar a cabo las pruebas.

1. CARACTERÍSTICAS INTEL CORE I5

- Procesador
  - Modelo: Intel Core I5 2500.
  - Frecuencia base: 3.3 GHz.
  - Frecuencia máxima: 3.7 GHz.
  - Número de procesadores: 1.
  - Número de cores (por procesador): 4.
  - Número de threads (por procesador): 4.
  - Conjunto de instrucciones: 64 bit.
  - Extensiones del conjunto de instrucciones: SSE4.1/4.2, AVX.
  - Datasheet:http://www.intel.com/content/www/us/en/processors/core/CoreTechnicalResources.html
- Memoria
  - Principal
    - Cantidad: 4 GB.
    - Frecuencia 1333 MHz.
    - Anchura: 64 bits.
    - Número de bancos: 1.
  - Caché (por procesador)
    - Nivel 1 (L1): 32x4 KB.
    - Nivel 2 (L2): 256x4 KB.
    - Nivel 3 (L3): 6 MB.

2. CARACTERÍSTICAS INTEL CORE I7

- Procesador
  - Modelo: Intel Core I7 2600.
  - Frecuencia base: 3.4 GHz.
  - Frecuencia máxima: 3.8 GHz.
  - Número de procesadores: 1.
  - Número de cores (por procesador): 4.
  - Número de threads (por procesador): 8.
  - Conjunto de instrucciones: 64 bit.
  - Extensiones del conjunto de instrucciones: SSE4.1/4.2, AVX.
  - Datasheet:http://www.intel.com/content/www/us/en/processors/core/CoreTechnicalResources.html
- Memoria
  - Principal
    - Cantidad: 8 GB.
    - Frecuencia 1333 MHz.
    - Anchura: 64 bits.
    - Número de bancos: 1.
  - Caché (por procesador)
    - Nivel 1 (L1): 32x4 KB.
    - Nivel 2 (L2): 256x4 KB.
    - Nivel 3 (L3): 8 MB.

3. CARACTERÍSTICAS DE MÁQUINA CC – NUMA

- Procesador
  - Modelo: Intel XEON E7-4807.
  - Frecuencia base: 1.87 GHz.
  - Frecuencia máxima: 1.87 GHz.
  - Número de procesadores: 4.
  - Número de cores (por procesador): 6.
  - Número de threads (por procesador): 12.
  - Conjunto de instrucciones: 64 bit.
  - Extensiones del conjunto de instrucciones: SSE4.1/4.2.
  - Datasheet:http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family/XeonE7TechnicalResources.html
- Memoria
  - Principal
    - Cantidad: 32x4 GB.
    - Frecuencia 1333 MHz.
    - Anchura: 64 bits.
    - Número de bancos: 4.
  - Caché (por procesador)
    - Nivel 1 (L1): 32x6 KB.
    - Nivel 2 (L2): 256x6 KB.
    - Nivel 3 (L3): 18 MB.

ANEXO II – INSTALACIÓN DE LIBRERÍAS Y HERRAMIENTAS

En este anexo se detalla el proceso de instalación de las librerías y herramientas que se han utilizado en la elaboración del proyecto.

1. VALGRIND Y KCACHEGRIND

Valgrind y Kcahegrind se encuentra en los repositorios oficiales de Ubuntu, por lo que su instalación se puede realizar a través de los mismos. Para llevar a cabo la instalación de Valgrind debe ejecutarse la siguiente línea en la consola de Linux:

```
sudo apt-get install valgrind
```

Para la instalación de Kcachegrind la siguiente:

```
sudo apt-get install kcachegrind
```

En el caso en que no se encuentre en los repositorios o se utilice otra distribución Linux Valgrind se puede descargar desde http://valgrind.org/downloads/current.html. Para su instalación se deben seguir los siguientes pasos:

- Descomprimir el fichero `tar.bz2` descargado con el comando `tar -xjvf fichero.tar.bz2`.
- Ejecutar, en la carpeta donde se descomprimieron los ficheros, el comando `./configure`.
- Compilar el código con el comando `make`.
- Instalar el programa con el comando `make install`.

Para la instalación de Kcachegrind fuera de repositorios se puede descargar desde http://kcachegrind.sourceforge.net/html/Download.html. Una vez descargado se siguen los siguientes pasos:

- Descomprimir el fichero tgz descargado con el comando `tar -xvzf fichero.tgz`.
- Ejecutar el comando `cmake .` para generar el Makefile correspondiente.
- Ejecutar `make` para compilar los fichero fuente descargados.
- Ejecutar `make install` para instalar el programa.

2. THREADING BUILDING BLOCKS (TBB)

TBB se encuentra en los repositorios oficiales de Ubuntu hasta la versión 2.0, sin embargo para versiones posteriores es necesario descargarlo de la página de Intel.

La versión de TBB que se ha utilizado en este proyecto es la 4.0 update 4, que en el momento de comienzo era la última versión estable de la librería, la descarga de esta versión puede hacerse desde http://threadingbuildingblocks.org/file.php?fid=77.

Para la instalación de la biblioteca deben seguirse los siguientes pasos:

- Se descarga la versión para Linux de la biblioteca. Esta versión ya esta compilada, aunque se puede descargar el código fuente y compilarlo.
- Se descomprime el fichero descargado. Para este ejemplo vamos a suponer que se descomprime en la carpeta `/opt/intel`.
- En la carpeta `/opt/intel` se crea un enlace simbólico llamado `tbb` que apunte a la carpeta raíz de TBB, de esta forma si actualizamos la versión de `tbb` bastará con apuntar este enlace a la carpeta raíz de la nueva versión.
- Entramos en la carpeta `/opt/intel/tbb/bin` y se edita el script `tbbvars.sh`. La edición se realiza en la línea donde se encuentra la sentencia "`export TBBROOT=`" donde se modifica por "`export TBBROOT=/opt/intel/tbb`", es decir la carpeta raíz de TBB.
- Una vez modificado el script se edita el fichero `.bashrc`. Este fichero se encuentra en la carpeta personal del usuario y se ejecuta cada vez que el usuario inicia sesión. Al final del fichero se añade la siguiente línea: `source /opt/intel/tbb /bin/tbbvars.sh <arquitectura>`. El parámetro <arquitectura> indica el tipo de arquitectura de la máquina, los valores que puede tomar son *ia32, ia64* o *intel64*. De esta forma cada vez que el usuario inicie sesión se inicializarán las variables de entorno necesarias y la biblioteca estará lista para su uso.
- Finalmente para utilizar TBB sólo es necesario incluir los ficheros de cabecera en el código fuente que se desarrolle.

3. ARRAY BUILDING BLOCKS (ARBB)

Al igual que TBB, ArBB no se encuentra en los repositorios oficiales de Ubuntu, por lo que es necesario llevar a cabo una instalación manual.

La instalación de ArBB es muy similar a la de TBB ya que se descarga compilado y sólo es necesario ajustar las variables de entorno al igual que con TBB.

La descarga se realiza desde http://software.intel.com/en-us/articles/download-intel-array-building-blocks/. Los pasos a seguir para la instalación son los siguientes:

- Se descarga la versión para Linux de la biblioteca. Esta versión ya esta compilada, aunque se puede descargar el código fuente y compilarlo.
- Se descomprime el fichero descargado. Para este ejemplo vamos a suponer que se descomprime en la carpeta `/opt/intel`.
- En la carpeta `/opt/intel` se crea un enlace simbólico llamado `arbb` que apunte a la carpeta raíz de ArBB, de esta forma si actualizamos la versión de ArBB bastará con apuntar este enlace a la carpeta raíz de la nueva versión.
- Entramos en la carpeta `/opt/intel/arbb/tools/` y se edita el script `arbbvars.sh`. La edición se realiza en la línea donde se encuentra la sentencia "`export ARBB_ROOT=`" donde se modifica por "`export ARBB_ROOT=/opt/intel/arbb`", es decir la carpeta raíz de ArBB.
- Una vez modificado el script se edita el fichero `.bashrc`. Este fichero se encuentra en la carpeta personal del usuario y se ejecuta cada vez que el usuario inicia sesión. Al final del fichero se añade la siguiente línea: `source /opt/intel/arbb/tools/arbbvars.sh <arquitectura>`. El parámetro <arquitectura> indica el tipo de arquitectura de la máquina, los valores que puede tomar son *ia32, ia64* o *intel64*. De esta forma cada vez que el usuario inicie sesión se inicializarán las variables de entorno necesarias y la biblioteca estará lista para su uso.
- Finalmente para utilizar ArBB sólo es necesario incluir los ficheros de cabecera en el código fuente que se desarrolle.

4. LIKWID

Likwid es un software de código abierto que no se encuentra en los repositorios de Ubuntu. Se puede descargar de:

http://code.google.com/p/likwid/downloads/detail?name=likwid-2.3.0.tar.gz.

Una vez descargado se siguen los siguientes pasos:

- Se descomprime el fichero tar.gz con el comando `tar -czfv fichero.tar.gz`. En este ejemplo el directorio de descompresión será `/home/usuario/likwid`.
- Una vez descomprimido se procede a compilarlo. Para ello se utiliza el comando make en la carpeta donde se he descomprimido el software.
- Una vez compilado, en la misma carpeta, se ejecuta el comando make install para instalarlo.
- Para poder ejecutar `likwid-perfctr` es necesario tener instalado el modulo de kernel de Linux msr. En caso de que no esté activado se debe abrir una terminal de root y ejecutar `modprobe msr`.

Para utilizar likwid debe tenerse en cuenta que este consta de varias aplicaciones. En el presente proyecto se ha utilizado `likwid-perfctr` que permite consultar los contadores hardware de la máquina donde se ejecuta.

Para utilizar `likwid-perfctr` deben comprobarse primero que grupos de datos están disponibles en la máquina. Par esto ejecutamos, en la carpeta de instalación de likwid la línea sudo `likwid-perfctr -a`. Esta línea nos da información sobre que grupos existen en la máquina, ya que cada procesador tiene unos contadores.

Posteriormente para tomar los datos de un grupo se ejecuta `likwid-perfctr -g <grupo> -c <procesadores> ./<programa a ejecutar>`

Con esta línea se ejecuta `likwid-perfctr` sobra un programa de usuario definido con el parámetro `<programa a ejecutar>`. Con la opción `-g` se indica el grupo sobre el que se quieren tomar los datos. Con la opción `-c` se indica los procesadores de los cuales se quieren tomar los datos, suponiendo que existan varios.

ANEXO III – PLANIFICACIÓN

En este anexo se muestra la planificación del proyecto. Las figuras de las siguientes páginas muestran el diagrama de Gantt que surge de esta planificación. Este diagrama se ha dividido en 5 partes para su correcta visualización en el documento.

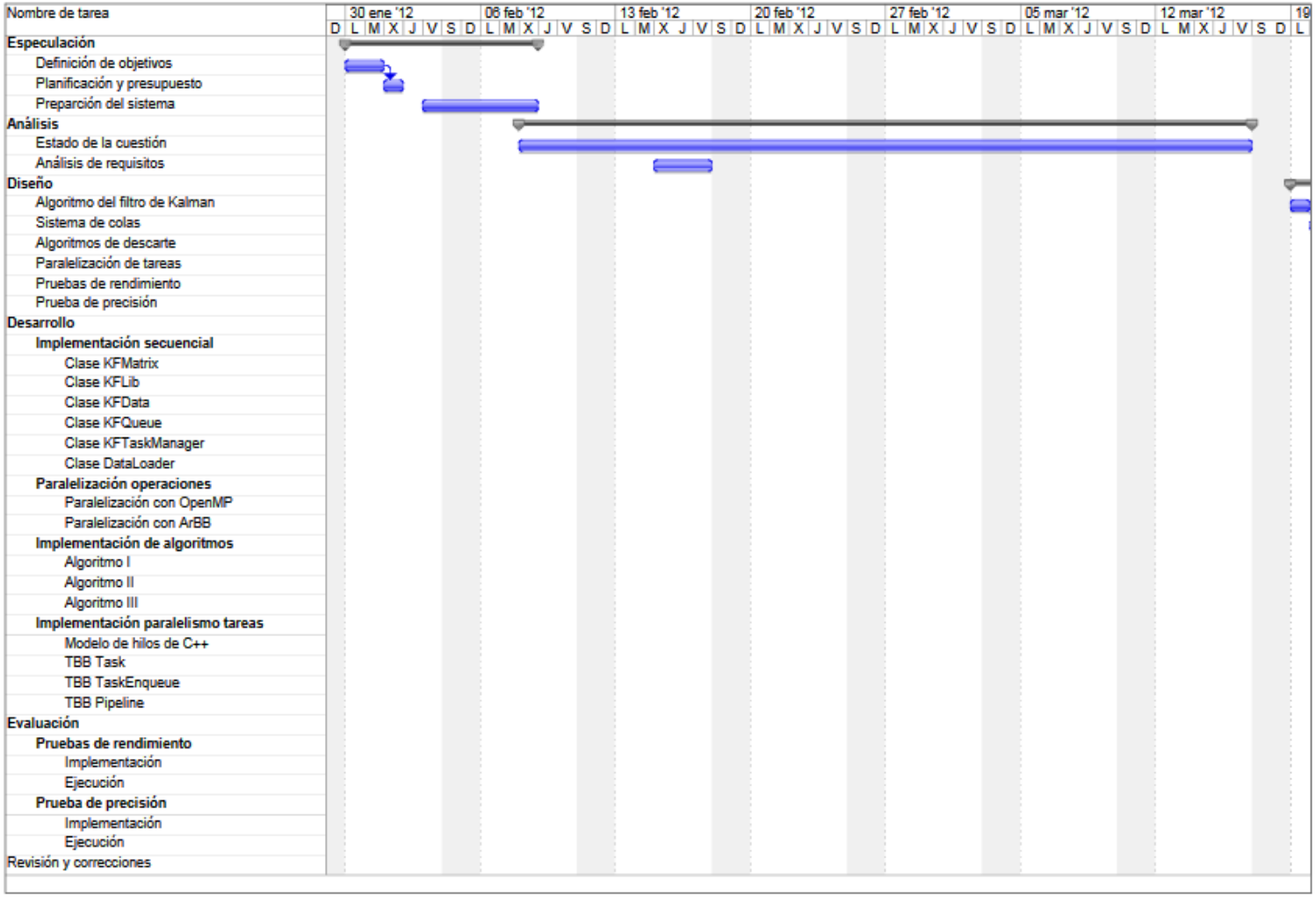

Figura Anexo II. Planificación I.

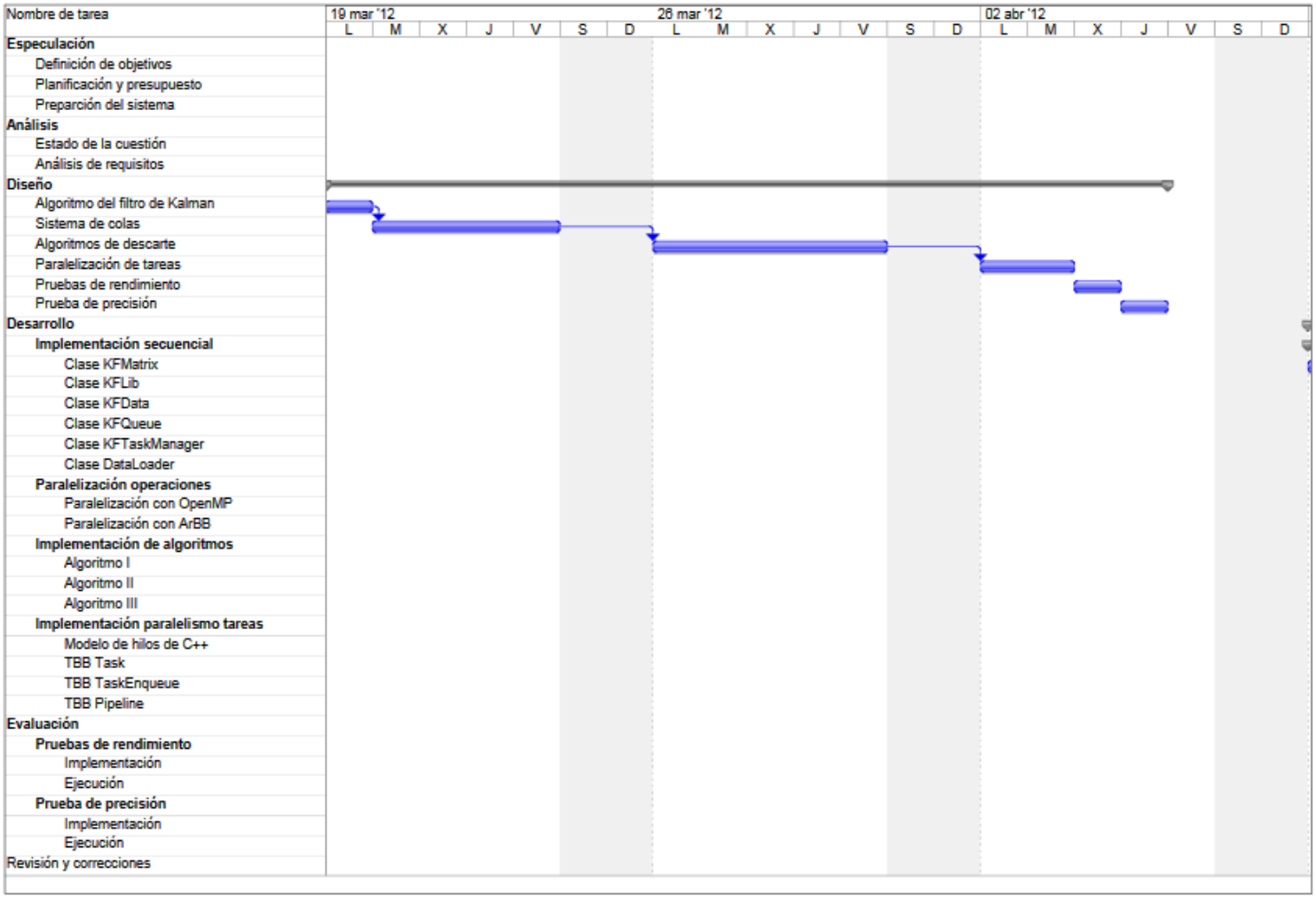

Figura Anexo II. Planificación II.

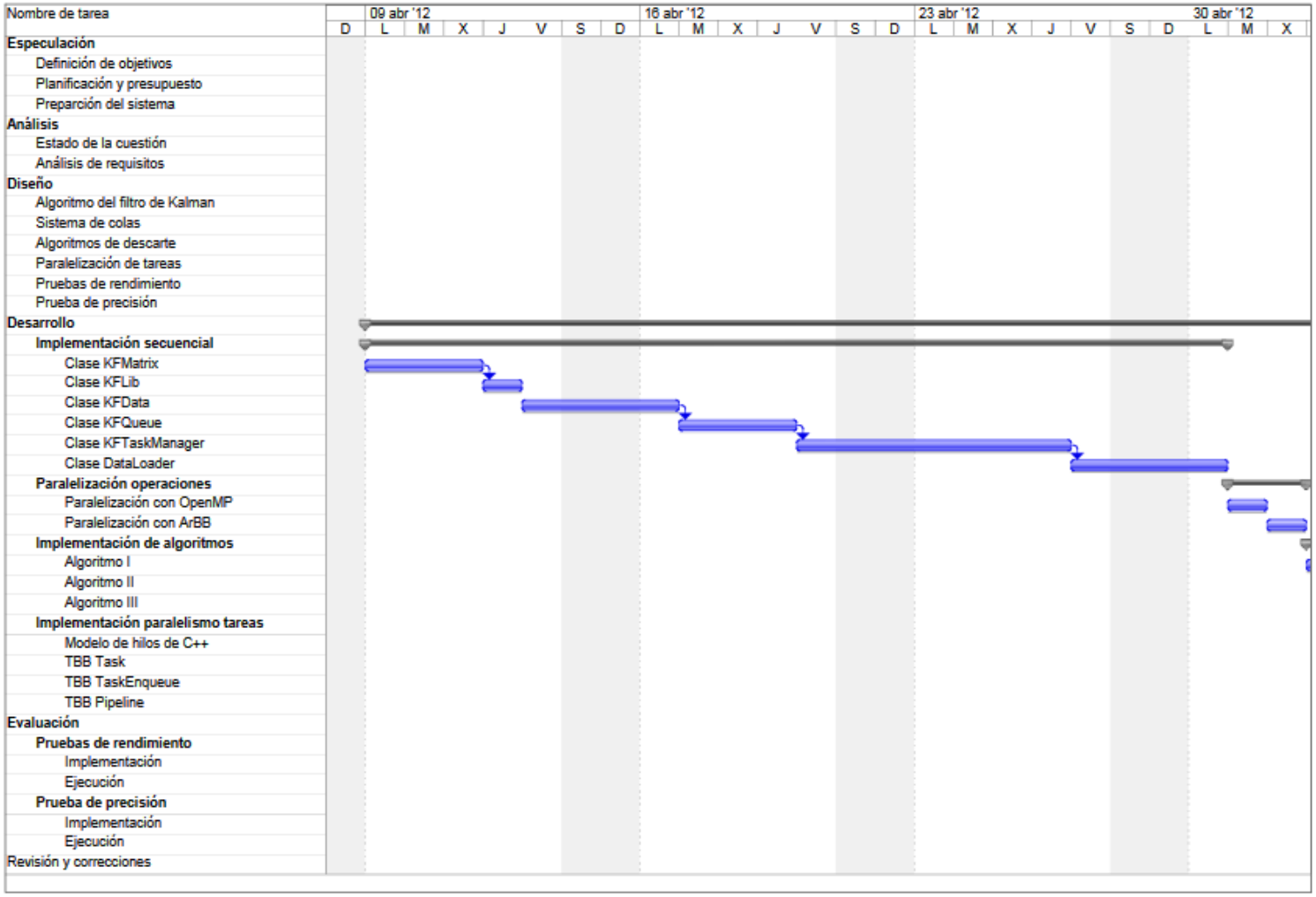

Figura Anexo II. Planificación III.

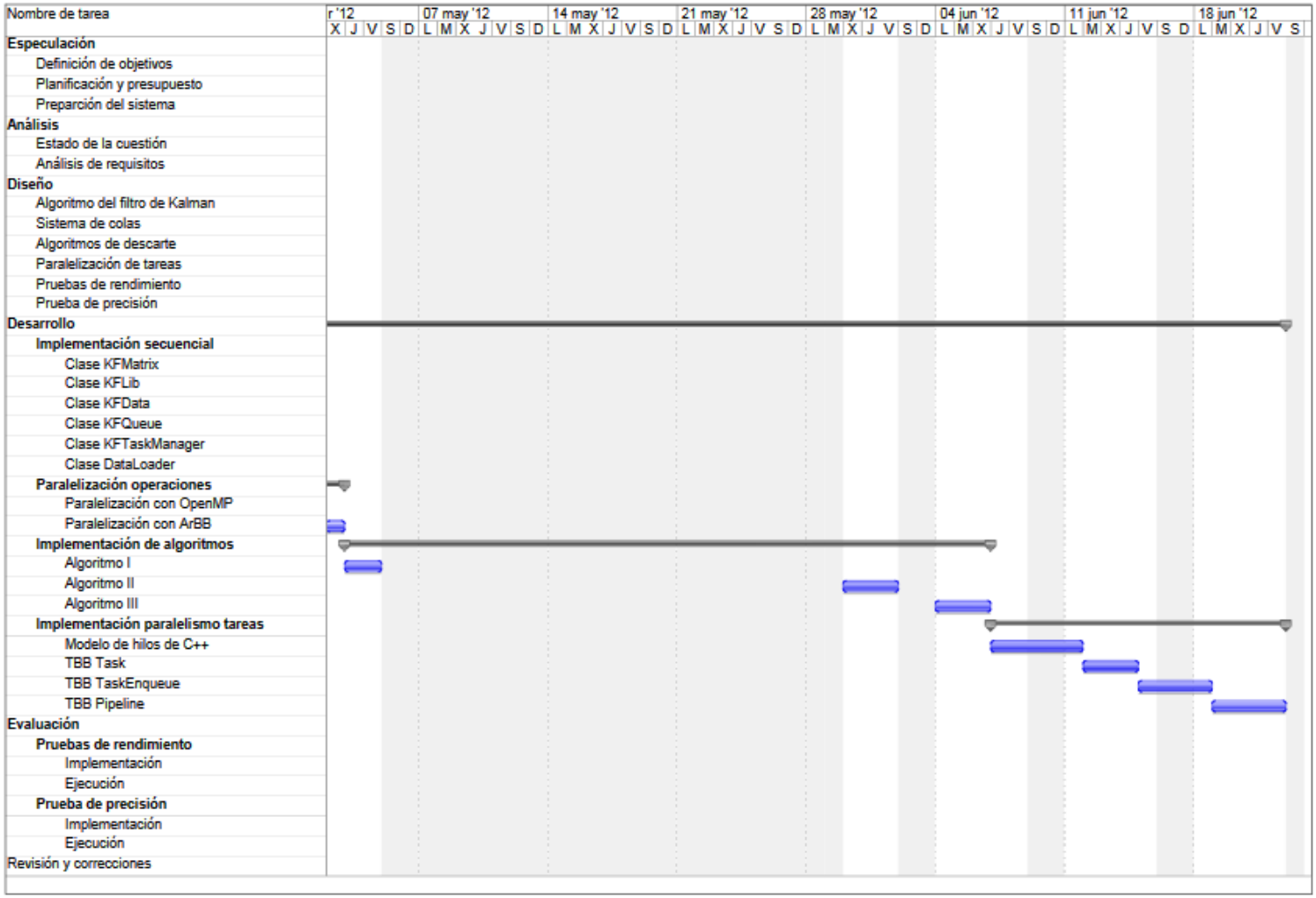

Figura Anexo II. Planificación IV.

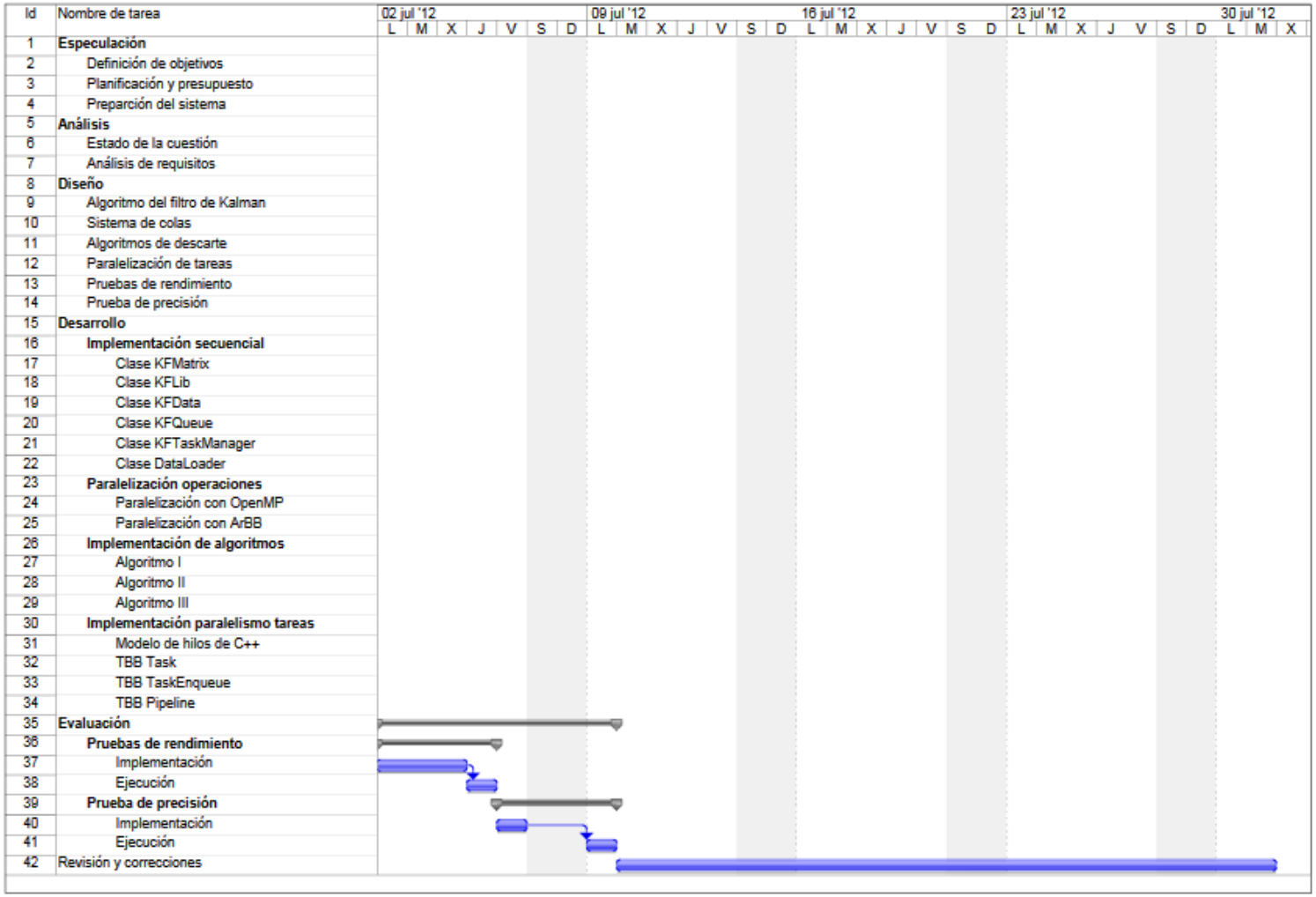

Figura Anexo II. Planificación V.

ANEXO IV - PRESUPUESTO

1.- Autor:

Javier Rodríguez Arroyo

2.- Departamento:

Departamento de Informática

3.- Descripción del Proyecto:

- Titulo **Uso de técnicas de paralelización para el algoritmo del filtro de Kalman.**
- Duración (meses) **6**

Tasa de costes Indirectos: **20%**

4.- Presupuesto total del Proyecto (valores en Euros):

14.268 Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar - solo a título informativo)	Categoría	Dedicación (hombres mes) [a)]	Coste hombre mes	Coste (Euro)
Javier Rodriguez Arroyo		Analista	2	3.000,00	6.000,00
Javier Rodriguez Arroyo		Diseñador	0,85	2.200,00	1.870,00
Javier Rodriguez Arroyo		Programador	2,3	1.300,00	2.990,00
Javier Rodriguez Arroyo		Ingeniero de pruebas	0,36	2.694,39	969,98
		Hombres mes	**5,51**	**Total**	11.829,98

[a)] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d)]
Maquina Intel Core 2 Duo	600,00	100	6	60	60,00
				Total	60,00

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} x C x D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	11.830
Amortización	60
Subcontratación de tareas	0
Costes de funcionamiento	0
Costes Indirectos	2.378
Total	14.268